ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «REST API система Service Desk з використанням мікросервісної архітектури»

на здобуття освітнього ступеня бакалавра
зі спеціальності 126 Інформаційні системи та технології
освітньо-професійної програми Інформаційні системи та технології

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

Вадим СТРЕЛЬБІЦЬКИЙ
(підпис) Ім'я, ПРІЗВИЩЕ здобувача

Виконав: здобувач вищої освіти гр. ІСД-41
Вадим СТРЕЛЬБІЦЬКИЙ
Ім'я, ПРІЗВИЩЕ

Керівник: Ірина СРІБНА
д-р техн. наук, доцент
Ім'я, ПРІЗВИЩЕ

Рецензент:
к.т.н., доцент
Ім'я, ПРІЗВИЩЕ

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інформаційних систем та технологій
Ступінь вищої освіти бакалавр
Спеціальність 126 Інформаційні системи та технології
Освітньо-професійна програма Інформаційні системи та технології

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІСТ
Каміла СТОРЧАК
«____» _____2026р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Стрельбіцькому Вадиму Юрійовичу

(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: REST API система Service Desk з використанням мікросервісної архітектури міським освітленням на базі IoT-технологій
керівник кваліфікаційної роботи Ірина СРІБНА, д-р техн. наук, доцент
(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 р. № 51
2. Строк подання кваліфікаційної роботи «01» червня 2026 р.

3. Вихідні дані кваліфікаційної роботи:
1. Основи розробки REST API та мікросервісної архітектури.
2. Вимоги до Service Desk систем та процесів обробки звернень користувачів.
3. Матеріали практичної реалізації REST API системи Service Desk, науково-технічна література.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
1. Аналіз теоретичних засад розробки REST API системи Service Desk.
2. Проєктування архітектури, моделі даних та REST API-контрактів системи.
3. Практична реалізація та оцінювання ефективності REST API системи Service Desk.

5. Перелік ілюстративного матеріалу: архітектурні схеми системи, таблиці API-маршрутів, результати тестування.
6. Дата видачі завдання «20» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Підбір та аналіз технічної літератури	20.03-27.03.26	
2.	Аналіз Service Desk, REST API та мікросервісної архітектури	28.03-06.04.26	
3.	Проєктування архітектури, моделі даних та API-контрактів	07.04-17.04.26	
4.	Розробка практичної частини REST API системи Service Desk	18.04-05.05.26	
5.	Тестування, аналіз ефективності та підготовка висновків	06.05-16.05.26	
6.	Розробка демонстраційних матеріалів, доповідь	17.05-22.05.26	
7.	Оформлення бакалаврської роботи	23.05-30.05.26	

Здобувач(ка) вищої освіти _____ Вадим СТРЕЛЬБІЦЬКИЙ
(підпис) (Ім’я, ПРІЗВИЩЕ)

Керівник кваліфікаційної роботи _____ Ірина СРІБНА
(підпис) (Ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 71 стор., 6 рис., 12 табл., 20 джерел.

Мета роботи - розробити та дослідити REST API систему Service Desk з використанням мікросервісної архітектури.

Об'єкт дослідження - процеси реєстрації, обробки та контролю виконання звернень користувачів у системі Service Desk.

Предмет дослідження - архітектурні рішення, REST API-контракти, моделі даних та програмні засоби реалізації Service Desk на основі мікросервісів.

Методи дослідження - системний аналіз, порівняння архітектурних підходів, моделювання предметної області, проєктування REST API, програмна реалізація прототипу, функціональне тестування та аналіз результатів роботи системи.

Короткий зміст роботи. У кваліфікаційній роботі обґрунтовано актуальність створення Service Desk системи для централізованої обробки звернень користувачів. Проаналізовано теоретичні засади REST API, мікросервісної архітектури, безпеки та спостережуваності сервісних систем. Спроєктовано предметну область, модель даних, життєвий цикл заявки та REST API-контракти. Практична частина присвячена реалізації прототипу на основі FastAPI, PostgreSQL, Docker Compose та Nginx, а також перевірці функціональних сценаріїв, контролю доступу, продуктивності й напрямів подальшого розвитку. [5] [7] [8] [10]

КЛЮЧОВІ СЛОВА: SERVICE DESK, REST API, МІКРОСЕРВІСНА АРХІТЕКТУРА, ЗАЯВКА, SLA, API GATEWAY, АВТЕНТИФІКАЦІЯ, DOCKER, POSTGRESQL, FASTAPI.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API - прикладний програмний інтерфейс

REST - архітектурний стиль взаємодії у веб-системах

HTTP - протокол передавання гіпертексту

JSON - текстовий формат обміну структурованими даними

JWT - маркер доступу для автентифікації користувача

SLA - угода про рівень сервісу

SQL - мова структурованих запитів до бази даних

ORM - об’єктно-реляційне відображення даних

MVP - мінімально життєздатний програмний продукт

UI - користувацький інтерфейс

ВСТУП

Актуальність теми дослідження зумовлена тим, що сучасні підприємства дедалі більше залежать від стабільної роботи інформаційних систем, внутрішніх сервісів, мережевої інфраструктури та засобів цифрової комунікації. У таких умовах звернення користувачів до служби підтримки повинні оброблятися не випадково, а за формалізованим процесом із чітким контролем строків, відповідальних осіб, статусів і результатів виконання. Саме тому Service Desk виступає важливим інструментом організації технічної підтримки, управління інцидентами та забезпечення безперервності бізнес-процесів.

Традиційні підходи до обліку звернень, які ґрунтуються на електронній пошті, телефонних повідомленнях або неструктурованих таблицях, не забезпечують достатнього рівня прозорості, аналітики та керованості. У таких умовах частина заявок може втрачатися, строки виконання складно контролювати, а керівництво не отримує повної інформації про якість роботи служби підтримки. Впровадження спеціалізованої Service Desk системи дозволяє централізувати процеси реєстрації, класифікації, маршрутизації та контролю виконання звернень користувачів.

Особливе значення у розробці такої системи має архітектура програмного рішення. Використання REST API забезпечує уніфікований механізм взаємодії між клієнтським інтерфейсом, серверними компонентами та можливими зовнішніми сервісами. Мікросервісна архітектура, у свою чергу, дозволяє розділити систему на окремі функціональні модулі, зменшити зв’язність між ними, підвищити масштабованість і спростити подальший розвиток програмного продукту.

Метою кваліфікаційної роботи є розробка та дослідження REST API системи Service Desk з використанням мікросервісної архітектури, яка забезпечує реєстрацію користувачів, автентифікацію, створення та обробку заявок, коментування, контроль пріоритетів, базові SLA-показники та взаємодію компонентів через стандартизований прикладний інтерфейс.

Об’єктом дослідження є процеси реєстрації, обробки та контролю виконання

звернень користувачів у системі Service Desk. Предметом дослідження є архітектурні рішення, REST API-контракти, моделі даних та програмні засоби реалізації Service Desk на основі мікросервісного підходу.

Для досягнення поставленої мети необхідно вирішити такі завдання: проаналізувати роль Service Desk у сучасній інформаційній інфраструктурі підприємства; дослідити принципи побудови REST API та мікросервісної архітектури; визначити функціональні й нефункціональні вимоги до системи; спроєктувати модель даних, життєвий цикл заявки та API-контракти; реалізувати практичний прототип системи; перевірити роботу основних сценаріїв і визначити напрями подальшого розвитку.

У роботі використано методи системного аналізу для визначення вимог і структури предметної області, методи об’єктно-орієнтованого та архітектурного проєктування для побудови моделі системи, а також практичні методи програмної реалізації, тестування REST API та контейнеризації сервісів. Практична частина базується на використанні FastAPI, PostgreSQL, Docker Compose, Nginx, JWT-автентифікації та клієнтського веб-інтерфейсу.

Практичне значення роботи полягає у створенні демонстраційного прототипу Service Desk системи, який може бути використаний як основа для подальшого розширення до повноцінної корпоративної платформи підтримки користувачів. Запропонована структура дозволяє поступово додавати сервіси сповіщень, аналітики, повнотекстового пошуку, інтеграції з корпоративним каталогом користувачів та іншими інформаційними системами організації.

Кваліфікаційна робота складається зі вступу, трьох розділів основної частини, висновків та переліку посилань. У першому розділі розглянуто теоретичні засади розробки REST API системи Service Desk. У другому розділі здійснено проєктування архітектури, моделі даних та REST API-контрактів. У третьому розділі описано практичну реалізацію, тестування, оцінювання продуктивності та напрями розвитку системи.

1 ТЕОРЕТИЧНІ ЗАСАДИ РОЗРОБКИ REST API СИСТЕМИ SERVICE DESK З ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

1.1 Сутність, роль і місце Service Desk у сучасній інформаційній інфраструктурі підприємства

У сучасних організаціях інформаційні технології перестали бути допоміжним інструментом і фактично стали основою щоденної діяльності. Від стабільної роботи робочих станцій, корпоративних сервісів, мережевої інфраструктури, систем електронного документообігу, засобів комунікації та внутрішніх веб-платформ залежить не лише комфорт персоналу, а й безперервність бізнес-процесів. Саме тому зростає значення систем, які дозволяють централізовано приймати, класифікувати, контролювати та супроводжувати звернення користувачів. У цьому контексті Service Desk виступає не просто каналом для фіксації технічних проблем, а організаційним і програмним механізмом, що формалізує взаємодію між користувачем та службою підтримки. Через нього проходять інциденти, сервісні запити, запити на доступ, консультації, повідомлення про збої та пропозиції щодо покращення якості обслуговування.

На відміну від спрощеного підходу, де підтримка зводиться лише до реакції на помилки, сучасний Service Desk орієнтується на надання послуги як керованого процесу. Якщо help desk здебільшого концентрується на швидкому усуненні окремої проблеми, то service desk охоплює ширший цикл: прийняття звернення, уточнення контексту, визначення пріоритету, контроль строків реагування, ескалацію, інформування заявника, накопичення знань та формування аналітики для управлінських рішень. Завдяки цьому керівництво отримує можливість не лише бачити поточний стан навантаження на команду підтримки, а й виявляти системні вузькі місця: повторювані інциденти, проблемні підрозділи, ресурси з найвищою кількістю збоїв, типові помилки користувачів, перевантаження окремих спеціалістів або порушення цільових показників обслуговування.

Практична цінність Service Desk особливо помітна в середовищах, де

кількість звернень є великою, а самі процеси повинні бути відтворюваними та прозорими. Без єдиного механізму реєстрації звернення нерідко надходять через телефонні дзвінки, електронну пошту, месенджери або усні повідомлення. У такому випадку частина запитів втрачається, а строк їх виконання залежить не від регламенту, а від людського фактора. Коли ж організація переходить до централізованої системи Service Desk, кожне звернення отримує унікальний ідентифікатор, статус, категорію, час створення, виконавця та історію змін. Це суттєво зменшує операційний хаос і дає можливість розглядати підтримку не як набір розрізнених дій, а як керований сервіс із чіткими параметрами якості.

З погляду бізнесу Service Desk виконує кілька критично важливих функцій. По-перше, він забезпечує єдину точку входу для всіх звернень, що спрощує комунікацію для кінцевого користувача. По-друге, він підтримує регламентну обробку запитів, коли правила маршрутизації, призначення відповідального, обчислення пріоритету та контроль строків реалізуються автоматично. По-третє, він формує цілісну картину щодо стану IT-послуг та пов’язаних із ними проблем. По-четверте, він створює історичну базу, яка може бути використана для аналізу якості сервісу, планування ресурсів, оцінки ефективності підрозділів підтримки та підготовки управлінської звітності. Врешті-решт, Service Desk виступає елементом зрілого IT-управління, оскільки дозволяє пов’язати технічну підтримку з реальними потребами організації.

Однією з ключових характеристик Service Desk є орієнтація на життєвий цикл звернення. Користувач не просто повідомляє про проблему, а ініціює процес, у межах якого відбувається послідовна зміна станів. Типовий цикл передбачає створення звернення, його первинний аналіз, визначення категорії та пріоритету, призначення виконавця, виконання робіт, погодження з користувачем результату та закриття звернення. У реальних системах цей життєвий цикл може бути складнішим: передбачати повторне відкриття, перенаправлення між групами, погодження змін, прикріплення документів, фіксацію фактично витраченого часу, контроль порушення SLA, формування внутрішніх нотаток та автоматичні сповіщення. Таким чином, Service Desk є не просто базою заявок, а процесним

середовищем для управління підтримкою.

Для великих організацій Service Desk часто виходить за межі IT-підтримки в класичному розумінні. Подібні системи можуть використовуватися службами безпеки, адміністративними відділами, відділом кадрів, бухгалтерією, юридичною службою або внутрішнім сервісним центром. Це означає, що програмна платформа повинна бути достатньо гнучкою, аби підтримувати різні типи звернень, різні маршрути обробки, окремі правила доступу, шаблони форм, довідники послуг і незалежні черги. Саме тут виникає потреба не лише у фронтальній зручності інтерфейсу, а насамперед у стабільному прикладному інтерфейсі, який здатний забезпечити обмін даними між веб-клієнтом, мобільним застосунком, модулем звітності, сервісом автентифікації, системою сповіщень та іншими підсистемами.

У контексті цифрової трансформації особливу роль відіграє інтеграційний потенціал Service Desk. Якщо раніше подібна система могла існувати ізольовано, то нині від неї очікується взаємодія з корпоративним каталогом користувачів, електронною поштою, каналами чат-комунікації, системами моніторингу, інструментами керування активами, базою знань, аналітичними панелями та сервісами централізованої автентифікації. Щоб така екосистема працювала без дублювання даних і ручного перенесення інформації, основою має стати чітко спроєктований API-рівень. Саме REST API у цьому випадку перетворюється на сполучну ланку між бізнес-логікою системи та зовнішніми клієнтами.

Ще однією важливою причиною впровадження Service Desk є потреба у вимірюванні якості підтримки. Організація не може ефективно покращувати процеси, якщо не має достовірних даних про тривалість реагування, середній час вирішення, частку повторно відкритих звернень, розподіл заявок за категоріями, навантаження на окремі групи або частоту порушення угод про рівень сервісу. Service Desk дає змогу збирати такі дані автоматично, а отже переводить оцінку роботи з площини суб’єктивних вражень у площину вимірюваних показників. Для цього в програмній системі мають бути передбачені події, часові мітки, статусні переходи та механізми аудиту.

Не менш важливим є питання стандартизації. У зрілих організаціях

підтримка користувачів повинна спиратися не на особисту ініціативу окремих працівників, а на узгоджені правила. Вони визначають, які категорії звернень існують, хто має право створювати певні типи заявок, як встановлюється пріоритет, які строки є допустимими, коли необхідна ескалація, у яких випадках потрібне погодження керівника, а коли звернення може бути закрите автоматично. Усе це неможливо забезпечити надійно без інформаційної системи, яка зберігає правила, перевіряє вхідні дані та керує переходами між станами. Саме тому проєктування Service Desk доцільно починати не з дизайну інтерфейсу, а з формалізації доменної моделі та сервісних сценаріїв.

Таким чином, Service Desk у сучасному підприємстві виступає одночасно інструментом операційного керування, засобом стандартизації сервісних процесів, джерелом аналітичних даних і платформою інтеграції. Для реалізації цих функцій недостатньо простого монолітного застосунку з мінімальною логікою обробки заявок. Потрібна гнучка система, здатна адаптуватися до змін бізнес-процесів, масштабуватися разом із ростом кількості користувачів, підтримувати безпечний обмін даними та забезпечувати стабільну роботу навіть за високого навантаження. Саме тому перспективним підходом є розробка Service Desk як набору взаємодіючих сервісів, об’єднаних через REST API.

Таблиця 1.1

Порівняння підходів Help Desk та Service Desk

Критерій	**Help Desk**	**Service Desk**	**Практичний ефект**
Фокус роботи	Усунення окремих технічних проблем	Керування сервісним процесом та якістю обслуговування	Підвищення керованості IT-послуг
Облік звернень	Фрагментарний або локальний	Централізований журнал заявок та історії змін	Прозорість і відтворюваність
Аналітика	Обмежена	Розвинені метрики, SLA, звіти	Підтримка управлінських рішень
Інтеграція	Мінімальна	Взаємодія з каталогами, поштою, моніторингом, API	Єдина сервісна екосистема

Порівняльний аналіз демонструє, що сучасний Service Desk є не просто

засобом реєстрації проблем, а платформою сервісного керування, яка дозволяє формалізувати строки виконання, фіксувати історію взаємодії та накопичувати дані для подальшого вдосконалення підтримки.

1.2 REST API як технологічна основа взаємодії між компонентами системи

Під час проєктування сучасних інформаційних систем одним із центральних завдань є створення зрозумілого, стабільного та масштабованого механізму взаємодії між клієнтськими застосунками й серверною частиною. У випадку Service Desk цей аспект набуває особливої ваги, оскільки система має обслуговувати різні типи клієнтів: веб-інтерфейс для користувачів, окремий інтерфейс для агентів підтримки, панелі аналітики, фонові сервіси, модулі сповіщення, а інколи також мобільний застосунок або інтеграції із зовнішніми платформами. Використання REST API дозволяє сформувати єдиний контракт обміну даними, у межах якого всі клієнти працюють із однаковою моделлю ресурсів і єдиними правилами звернення до серверної логіки.

REST розглядається не як конкретний протокол або бібліотека, а як архітектурний стиль, що визначає принципи організації взаємодії в розподілених системах. Його практична привабливість пояснюється тим, що він добре поєднується з веб-технологіями, спирається на можливості HTTP і дозволяє будувати API, логічно орієнтований на ресурси. У Service Desk такими ресурсами можуть бути користувачі, ролі, заявки, коментарі, вкладення, категорії, статуси, черги, правила SLA, журнали подій і повідомлення. Кожен ресурс має унікальну адресу, підтримує обмежений набір операцій та повертає структуроване представлення даних у стандартизованому форматі, найчастіше JSON.

Перевага ресурсно-орієнтованого підходу полягає в тому, що він примушує розробника мислити в термінах сутностей предметної області, а не в термінах окремих процедур або технічних викликів. Якщо система будується навколо сутностей заявки, користувача, коментаря чи черги, то і прикладний інтерфейс набуває природної форми: створення заявки відбувається через POST-запит до

колекції заявок, отримання конкретної заявки - через GET-запит за її ідентифікатором, зміна стану - через відповідний запит оновлення, а отримання списку заявок із фільтрацією - через параметризований GET-запит. Така структура суттєво знижує складність інтеграції, оскільки поведінка API є передбачуваною й не потребує запам’ятовування великої кількості нестандартних операцій.

Однією з фундаментальних властивостей REST є безстанність взаємодії. Це означає, що кожен запит має містити всю необхідну інформацію для його обробки, а сервер не повинен покладатися на прихований контекст попередніх викликів. Для Service Desk така модель є вкрай корисною, тому що дозволяє масштабувати систему горизонтально. Якщо запити не залежать від локального стану окремого серверного вузла, їх можна розподіляти між кількома екземплярами сервісу без додаткових складних механізмів синхронізації. У результаті зростає відмовостійкість і спрощується розгортання системи у хмарному або контейнеризованому середовищі.

Практична реалізація REST API тісно пов’язана з використанням HTTP-методів. У рамках Service Desk метод GET застосовується для читання даних: отримання списку заявок, картки користувача, переліку категорій, історії змін або статистичних показників. Метод POST доцільно використовувати для створення нових ресурсів, зокрема нової заявки, коментаря чи службового повідомлення. Метод PUT або PATCH застосовується для повного чи часткового оновлення ресурсу, наприклад для зміни статусу заявки, призначення відповідального, оновлення профілю користувача чи параметрів SLA. Метод DELETE використовується рідше, оскільки в сервісних системах фізичне видалення даних зазвичай обмежується з міркувань аудиту, але він може бути доречним для допоміжних сутностей або логічного архівування.

Коректне використання статусних кодів HTTP має не менш важливе значення, ніж побудова адрес ресурсів. У добре спроєктованому API відповідь сервера повинна бути зрозумілою не лише для людини, а й для клієнтського застосунку, який автоматично реагуватиме на різні ситуації. Успішне отримання даних позначається кодом 200, успішне створення ресурсу - 201, прийняття задачі

до асинхронної обробки - 202, відсутність контенту у відповіді - 204. Помилки на боці клієнта повинні супроводжуватися кодами класу 4xx, а серверні помилки - кодами класу 5xx. Для Service Desk це важливо, оскільки клієнт повинен чітко розрізняти ситуації некоректного запиту, відсутності прав доступу, конфлікту станів, відсутності ресурсу або тимчасової внутрішньої несправності сервісу.

Окремої уваги заслуговує питання структури представлення даних. Формат JSON став де-факто стандартом для REST API завдяки компактності, читабельності та широкій підтримці в мовах програмування й клієнтських фреймворках. Для системи Service Desk це означає можливість уніфікувати схеми відповіді для різних ресурсів і забезпечити зручну валідацію на стороні клієнта. Наприклад, відповідь на запит списку заявок може містити масив об’єктів із полями ідентифікатора, теми, статусу, пріоритету, автора, призначеного агента, часових міток та ознаки порушення SLA. Водночас важливо уникати надмірної вкладеності та дублювання даних, оскільки це ускладнює клієнтську логіку і збільшує обсяг трафіку.

У реальних системах REST API має враховувати не лише базові CRUD-операції, а й низку практичних механізмів, без яких використання інтерфейсу буде незручним. До них належать пагінація, сортування, фільтрація, пошук, частковий вибір полів, масові операції, ідемпотентність окремих запитів та версіонування. Для Service Desk ці механізми є критично важливими, адже обсяг заявок з часом швидко зростає. Якщо API не підтримує пагінацію, будь-який запит до журналу звернень може стати надто важким. Якщо відсутнє фільтрування, агенту складно отримати лише активні заявки своєї групи. Якщо немає версіонування, зміна структури відповіді може порушити роботу клієнтів після оновлення сервера. Отже, якісне REST API передбачає не лише передачу даних, а й продуману організацію доступу до них.

Ще одна важлива характеристика REST API - його придатність до документування та автоматизованої перевірки. Для системи Service Desk, яку можуть використовувати не лише внутрішні клієнти, а й сторонні інтегратори, критично важливо мати формальний опис доступних кінцевих точок, параметрів запиту, структур відповіді, правил автентифікації та кодів помилок. Наявність такої

специфікації зменшує кількість помилок при інтеграції, спрощує тестування й забезпечує узгодженість між командами розробки фронтенду та бекенду. Крім того, формальний опис API дозволяє генерувати клієнтські SDK, створювати макети сервісів для тестування та автоматично перевіряти відповідність реалізації визначеному контракту.

Для Service Desk REST API є не лише технічним інтерфейсом, а фактично мовою, якою різні підсистеми домовляються про спільну роботу. Через нього відбувається створення звернень із веб-форми, передача подій у систему сповіщень, завантаження довідників для клієнтського інтерфейсу, перевірка ролей доступу, отримання статистики для панелей моніторингу, а також обмін повідомленнями між мікросервісами. Саме тому при проєктуванні API важливо зберігати баланс між універсальністю, простотою та виразністю. Занадто абстрактний інтерфейс ускладнює використання, а надто вузькоспеціалізований швидко втрачає гнучкість.

Отже, REST API є природною технологічною основою для побудови сучасної системи Service Desk. Він забезпечує уніфікований спосіб доступу до доменних сутностей, спрощує інтеграцію між компонентами, підтримує масштабування та дозволяє будувати архітектуру, у якій логіка підтримки відокремлюється від клієнтських застосунків. Саме тому подальше проєктування системи доцільно здійснювати, виходячи з того, що вся бізнес-логіка та взаємодія між сервісами мають бути представлені через стабільний і добре структурований REST API.

1.3 Особливості мікросервісної архітектури та доцільність її застосування для Service Desk

Архітектурний стиль системи визначає не лише спосіб розміщення програмних модулів, а й те, наскільки легко система буде розвиватися, масштабуватися та супроводжуватися в майбутньому. Традиційний монолітний підхід передбачає, що весь функціонал застосунку - робота з користувачами, заявками, ролями, сповіщеннями, звітами, файлами, довідниками - реалізується в межах одного процесу і зазвичай спирається на спільну базу даних. Така модель є

прийнятною для невеликих проєктів на ранніх етапах, однак у міру зростання системи вона починає створювати обмеження. Будь-яка зміна в одному модулі потребує повторного розгортання всього застосунку, залежності між частинами зростають, а локальний збій або витік ресурсів може вплинути на роботу всієї системи.

Мікросервісна архітектура пропонує інший підхід: система розбивається на набір відносно невеликих сервісів, кожен із яких відповідає за окрему бізнес-можливість і розгортається незалежно від інших. Для Service Desk це особливо доречно, оскільки доменна область природно поділяється на логічні підсистеми. Автентифікація та керування ролями мають свою окрему логіку, модуль заявок - власні правила життєвого циклу, сервіс коментарів і вкладень - свої механізми зберігання, модуль сповіщень - окремі канали доставки та шаблони повідомлень, а підсистема аналітики - специфічні запити та моделі агрегації. Якщо все це виділено в окремі сервіси, система стає гнучкішою і краще пристосовується до еволюції вимог.

Однією з головних причин переходу до мікросервісів є можливість незалежного масштабування. У реальному середовищі навантаження на різні функції Service Desk є неоднорідним. Наприклад, у пікові періоди різко зростає кількість операцій читання списків заявок або перегляду історії, тоді як модуль довідників залишається майже статичним. Сервіс сповіщень може відчувати пікові навантаження в моменти масових інцидентів, а сервіс автентифікації - на початку робочого дня. Якщо архітектура монолітна, масштабувати доводиться весь застосунок цілком. Якщо ж система побудована як сукупність сервісів, можна збільшити кількість екземплярів лише тих компонентів, які реально зазнають підвищеного навантаження.

Важливою перевагою мікросервісного підходу є також технологічна та організаційна незалежність окремих компонентів. Різні сервіси можуть мати власні цикли змін, різні темпи розвитку і навіть різні вимоги до даних. Наприклад, для модуля заявок важлива консистентність транзакцій та швидке виконання пошукових запитів, тоді як для сервісу сповіщень на перший план виходить

асинхронна обробка черг і гарантована доставка повідомлень. Мікросервісна архітектура дозволяє уникати надмірної уніфікації там, де вона лише шкодить. Водночас така свобода повинна бути керованою, щоб система не перетворилася на набір різнорідних рішень без спільних правил.

Однак переваги мікросервісів стають реальними лише за умови грамотного визначення меж сервісів. Невдалий поділ системи призводить до надмірної кількості міжсервісних викликів, дублювання даних, складних сценаріїв узгодження змін та важких для підтримки ланцюжків залежностей. Тому для Service Desk доцільно виділяти сервіси не за технічними шарами, а за бізнес-контекстами. Один сервіс повинен відповідати за життєвий цикл заявок, інший - за користувачів та ролі, ще один - за надсилання повідомлень, окремий - за каталог послуг або базу знань. Такий підхід зменшує зв’язність і дозволяє кожному сервісу мати власну модель даних, не підлаштовану під внутрішні потреби інших модулів.

Суттєвим архітектурним питанням є спосіб комунікації між сервісами. У Service Desk частина взаємодій має синхронний характер. Наприклад, під час створення заявки сервіс заявок повинен переконатися, що користувач існує і має право створювати запит вибраного типу. Водночас багато подій доцільно обробляти асинхронно: надсилання електронного листа, запис в аудит, оновлення статистики, передача повідомлення в канал месенджера. Комбінація синхронного REST-виклику та асинхронних повідомлень через брокер подій дозволяє побудувати систему, де критичні перевірки виконуються одразу, а другорядні дії не блокують основний сценарій роботи користувача.

Для мікросервісної архітектури принциповим є питання зберігання даних. У зрілих реалізаціях кожний сервіс володіє власною базою або власною схемою даних і не залежить від прямого доступу до таблиць іншого сервісу. Це правило захищає від тісного зчеплення на рівні сховища. Для Service Desk воно означає, що сервіс користувачів керує власними сутностями користувачів та ролей, сервіс заявок - власними таблицями заявок і статусних переходів, сервіс сповіщень - чергами та журналом доставки, а сервіс аналітики - агрегованими зрізами або окремим сховищем для звітності. Обмін необхідними даними між сервісами

відбувається через API або події, а не через неформалізовані SQL-запити до чужих таблиць.

Разом із перевагами мікросервіси приносять і нові складнощі. Зростає кількість розгортань, конфігурацій, мережевих з’єднань, журналів, секретів доступу, точок відмови та сценаріїв деградації. Якщо моноліт можна досить швидко запустити на одному сервері, то мікросервісна система вимагає зрілого підходу до контейнеризації, автоматизації розгортання, централізованого логування, моніторингу, управління конфігураціями та відстеження запитів між сервісами. Для дипломного проєкту це означає, що архітектурне рішення має бути не лише теоретично привабливим, а й практично обґрунтованим. Якщо доменна область дійсно поділяється на самостійні контексти, а система розрахована на розвиток та інтеграції, мікросервіси є виправданим вибором.

Доцільність використання мікросервісів для Service Desk також пояснюється різною швидкістю зміни окремих модулів. Наприклад, бізнес-логіка заявок, маршрутизації та SLA зазвичай розвивається швидше, ніж базові механізми автентифікації. Сервіс сповіщень може потребувати розширення каналів доставки, тоді як модуль ролей залишається відносно стабільним. Незалежне розгортання дає змогу оновлювати необхідний сервіс без ризику випадково вплинути на решту системи. Це особливо важливо в середовищах, де доступність сервісу підтримки є критичною, а перерви в роботі мають бути мінімальними.

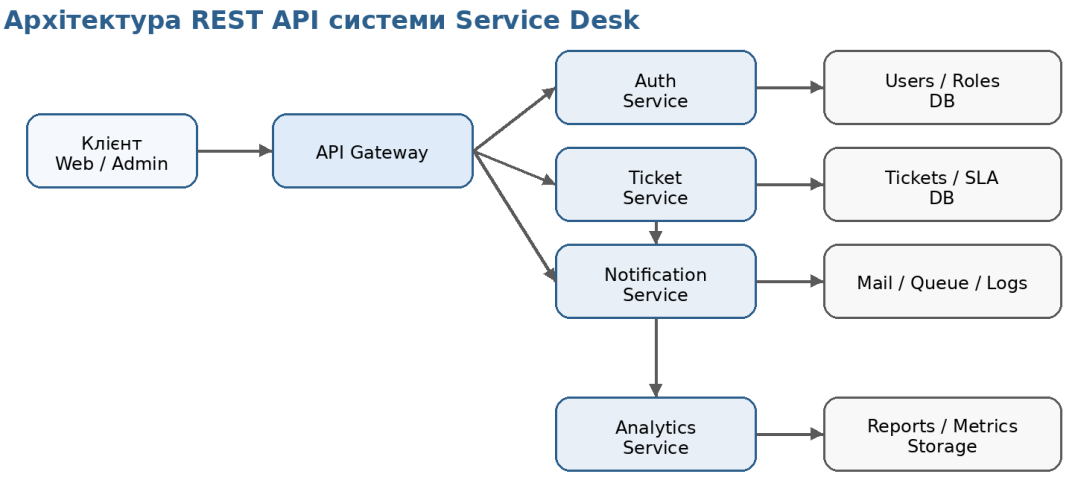


Рис. 1.1 Узагальнена архітектура взаємодії компонентів Service Desk

Наведена схема показує, що API Gateway виконує роль єдиної точки входу, а окремі мікросервіси беруть на себе власні бізнес-контексти. Такий підхід зменшує зв'язність між модулями, полегшує масштабування та дозволяє ізолювати зміни в межах конкретного сервісу без повного перевипуску всієї системи.

Не можна оминути і тему відмовостійкості. У мікросервісній архітектурі помилка в одному сервісі не обов'язково повинна призводити до повної недоступності системи. Якщо, наприклад, тимчасово недоступний модуль сповіщень, користувач усе одно може створити заявку, а самі повідомлення будуть доставлені пізніше після відновлення сервісу або через повторну обробку черги. Такий підхід вимагає окремих механізмів повторних спроб, тайм-аутів, захисту від каскадних відмов, однак у результаті система стає стійкішою до часткових збоїв. Для Service Desk, який часто використовується саме під час інцидентів, ця властивість має особливу цінність.

Отже, мікросервісна архітектура для системи Service Desk є доцільною насамперед тоді, коли від системи очікується масштабованість, інтегрованість, гнучкий розвиток функціоналу та можливість незалежно розвивати окремі бізнес-контексти. Разом із тим її застосування потребує дисципліни у визначенні меж сервісів, стандартизації API, продуманої моделі взаємодії, засобів

спостережуваності та контролю конфігурації. Саме ці аспекти мають бути враховані під час проєктування архітектури REST API системи Service Desk.

1.4 Функціональні та нефункціональні вимоги до сучасної системи Service Desk

Будь-яка інформаційна система повинна проектуватися не абстрактно, а виходячи з конкретних вимог, які визначають очікувану поведінку, обмеження та критерії якості. Для системи Service Desk формування таких вимог має особливе значення, тому що вона безпосередньо впливає на оперативну роботу персоналу, швидкість обслуговування користувачів та якість управлінської аналітики. Якщо на етапі проєктування не зафіксувати, які саме завдання повинна вирішувати система, надалі архітектура буде накопичувати випадкові рішення, а розробка перетвориться на реакцію на окремі запити без цілісної логіки. Тому доцільно розділити вимоги на функціональні та нефункціональні.

До основних функціональних вимог належить підтримка повного циклу роботи зі зверненнями. Система повинна забезпечувати створення заявок користувачами, перегляд картки звернення, оновлення статусу, зміну пріоритету, призначення виконавця, додавання коментарів, вкладень та службових нотаток, а також закриття або повторне відкриття заявки. При цьому важливо, щоб різні ролі бачили лише ті дії, які відповідають їхнім повноваженням. Користувач повинен мати змогу створювати й відстежувати власні звернення, агент підтримки - обробляти заявки своєї черги, керівник - отримувати аналітику та виконувати ескалації, адміністратор - керувати довідниками, ролями, шаблонами та правилами маршрутизації.

Окремою функціональною вимогою є підтримка класифікації звернень. У реальній експлуатації Service Desk не може обмежуватися лише текстовим описом проблеми. Заявка повинна бути пов’язана з категорією, підкатегорією, типом послуги, пріоритетом, джерелом створення та, за потреби, з певним активом або підрозділом. Така структура потрібна не лише для зручності обробки, а й для побудови звітності та автоматичних правил. Наприклад, заявки на доступ до

корпоративної системи можуть спрямовуватися в одну групу, інциденти мережевої інфраструктури - в іншу, а запити на зміну параметрів облікового запису - в третю. Без класифікації автоматизація швидко втрачає ефективність.

Ще однією важливою функцією є підтримка SLA та механізмів пріоритезації. Для Service Desk принципово важливо не лише фіксувати сам факт звернення, а й контролювати, у який строк воно має бути прийняте в роботу та вирішене. Це означає, що система повинна вміти обчислювати терміни реагування залежно від типу звернення, рівня критичності, робочого календаря, черги відповідальної групи та, за потреби, категорії користувача. Крім того, потрібні механізми попередження про ризик порушення строків, ескалації до старшого спеціаліста або керівника, а також відображення факту порушення в аналітичних звітах.

Функціонально важливою є і можливість ведення історії змін. У картці звернення повинно бути видно, хто створив заявку, коли вона була змінена, хто змінював статус, коли був призначений виконавець, які коментарі додавалися, коли надсилалися повідомлення та чи змінювався пріоритет. Така історія має значення як для внутрішнього контролю якості, так і для розв’язання спірних ситуацій. Якщо система не веде повноцінного аудиту, то згодом неможливо відтворити реальну картину подій, а це негативно впливає на довіру до сервісу.

Серед функціональних вимог також варто виділити пошук, фільтрацію та формування аналітичних зрізів. З часом у Service Desk накопичується значний обсяг даних, і без розвинених механізмів вибірки робота з ними стає неефективною. Користувачі повинні мати можливість знайти звернення за номером, темою, категорією, статусом, виконавцем, автором, періодом створення або іншими атрибутами. Керівники очікують агреговану інформацію: кількість нових заявок, середній час закриття, навантаження на групи, частку прострочених звернень, найпроблемніші категорії та інші показники. Отже, API системи має підтримувати як операційні сценарії, так і сценарії звітності.

Окрім функціональності, для системи Service Desk критичними є нефункціональні вимоги. Насамперед це продуктивність. Система повинна швидко обробляти типові запити навіть за умови, що кількість звернень і користувачів

зростає. Надмірні затримки під час відкриття списку заявок, переходу між сторінками або збереження коментарів безпосередньо знижують ефективність роботи агентів підтримки. Тому архітектура повинна враховувати індексацію, кешування, пагінацію, асинхронну обробку допоміжних задач та оптимізацію структури запитів до баз даних.

Таблиця 1.2

Узагальнення функціональних і нефункціональних вимог до системи Service Desk

Група вимог	**Конкретні вимоги**	**Вплив на архітектуру**
Функціональні	Створення, маршрутизація, коментування, закриття заявок	Потребує окремого сервісу заявок і ролей
Функціональні	SLA, пріоритети, ескалації, аудит подій	Необхідні часові правила й журналювання
Нефункціональні	Продуктивність, пагінація, фільтрація, кешування	Визначає структуру REST API та індексацію
Нефункціональні	Безпека, автентифікація, контроль доступу	Потребує окремого auth-сервісу і політик авторизації
Нефункціональні	Спостережуваність, резервування, масштабованість	Вимагає логування, метрик і розподіленого розгортання

Наступною нефункціональною вимогою є масштабованість. Навіть якщо на початковому етапі систему використовує обмежена кількість працівників, у перспективі вона може стати платформою для кількох підрозділів або навіть для зовнішніх користувачів. Це означає, що структура застосунку не повинна жорстко обмежувати зростання. Можливість масштабувати сервіси горизонтально, розподіляти навантаження, використовувати окремі сховища для різних контекстів і виносити частину задач у фонову обробку стає не опціональною перевагою, а практично необхідною вимогою.

Окремо слід розглядати вимоги до безпеки. Service Desk працює з персональними даними користувачів, внутрішньою службовою інформацією, даними щодо прав доступу, маршрутів погодження та технічного стану корпоративних сервісів. Витік або несанкціонована зміна таких даних є неприпустимими. Отже, система повинна забезпечувати надійну автентифікацію, розмежування прав доступу, захищену передачу даних, аудит дій, обмеження

частоти запитів, перевірку коректності вхідних даних та захист від поширених загроз, пов’язаних із API. Безпека має бути вбудована в архітектуру з самого початку, а не додаватися постфактум.

Нефункціональною вимогою є і спостережуваність системи. Оскільки мікросервісна архітектура передбачає розподілене виконання, команда супроводу повинна мати змогу швидко зрозуміти, де саме виникла помилка, який сервіс спрацював повільно, чому не було доставлено повідомлення або який компонент став джерелом деградації. Для цього потрібні централізовані журнали, метрики, трасування запитів, кореляційні ідентифікатори та механізми оповіщення про відхилення. Якщо система не є спостережуваною, її підтримка стає надто залежною від ручного аналізу та досвіду окремих розробників.

Важливою нефункціональною вимогою є також підтримуваність. Service Desk не є одноразовим програмним продуктом; це система, яка буде доповнюватися новими сценаріями, ролями, формами заявок, інтеграціями та звітами. Тому кодова база, API-контракти, правила валідації та конфігурації повинні бути організовані таким чином, щоб зміни не призводили до каскадних поломок. Саме з цієї причини мікросервісний підхід і добре визначені REST-контракти є раціональним вибором: вони дозволяють локалізувати зміни й зменшити ризик впливу на всю систему.

Отже, набір вимог до системи Service Desk охоплює не лише базову можливість створювати та закривати заявки. Йдеться про побудову платформи, що поєднує регламентну обробку звернень, контроль якості обслуговування, гнучке розмежування доступу, аналітику, інтеграцію з іншими сервісами та стабільну роботу за змінного навантаження. Саме такий підхід повинен лежати в основі архітектурних рішень, які розглядатимуться в наступному розділі.

1.5 Безпека, надійність та спостережуваність REST API системи Service Desk

Розробка прикладного інтерфейсу для Service Desk неможлива без урахування трьох взаємопов’язаних аспектів: безпеки, надійності та

спостережуваності. Якщо функціональність системи визначає, що саме вона робить, то ці характеристики визначають, наскільки безпечно, стабільно й контрольовано вона буде працювати в реальному середовищі. Для сервісних систем це особливо важливо, оскільки вони обслуговують критичні внутрішні процеси та накопичують чутливі дані про інциденти, доступи, зміни конфігурацій, службові коментарі та дії співробітників.

Питання безпеки починається з автентифікації та авторизації. У системі Service Desk не всі користувачі мають однакові права: звичайний співробітник може створювати запити і бачити лише власні звернення, агент підтримки отримує доступ до заявок своєї групи, керівник бачить аналітику та історію ескалацій, а адміністратор керує ролями, довідниками й системними параметрами. Отже, застосунок повинен не лише встановлювати особу користувача, а й перевіряти, які саме дії дозволені для кожного ресурсу. Для REST API це означає необхідність централізованого механізму перевірки маркерів доступу, ролей, дозволів та обмежень на рівні окремих операцій.

Серйозну загрозу для API становить недостатня перевірка доступу до конкретних об’єктів. Навіть якщо користувач успішно автентифікувався, він не повинен мати можливість довільно змінювати чужі заявки, бачити конфіденційні коментарі або редагувати системні довідники. Тому перевірка прав має виконуватися не лише на рівні маршруту, а й на рівні конкретного ресурсу та бізнес-правила. Наприклад, агент підтримки може змінювати статус заявки, яка належить його групі, але не повинен отримувати можливість переглядати внутрішні записи іншого підрозділу. У контексті Service Desk правильне розмежування доступу є ключовим для довіри до системи.

Не менш важливим є захист вхідних даних. API приймає текстові описи, параметри фільтрації, ідентифікатори, вкладення та інші дані, кожне з яких може стати джерелом помилки або зловживання. Некоректна валідація призводить не лише до аварійних ситуацій, а й до потенційних вразливостей. Саме тому в системі повинна бути реалізована перевірка форматів, обмежень довжини, списків допустимих значень, цілісності посилань на інші сутності та безпечної обробки

файлів. З практичної точки зору це означає, що жоден контролер REST API не повинен працювати без чітко визначеної схеми вхідних даних.

Надійність системи в мікросервісному середовищі означає здатність підтримувати працездатність навіть у разі часткових збоїв. Service Desk повинен залишатися доступним у моменти, коли він найбільш потрібний, тобто під час інцидентів, масових звернень або технічних проблем в інфраструктурі. Для цього необхідно передбачити тайм-аути на міжсервісних викликах, повторні спроби для нестабільних зовнішніх залежностей, ізоляцію фонових задач, захист від перевантаження черг і механізми деградації функціоналу. Наприклад, якщо сервіс сповіщень тимчасово недоступний, це не повинно блокувати створення заявки, хоча сама доставка повідомлення може бути відкладена.

Особливе значення має журналювання подій. Для системи Service Desk логи потрібні не тільки розробникам, а й адміністраторам та фахівцям з інформаційної безпеки. Вони дозволяють зрозуміти, хто звертався до конкретного ресурсу, які параметри були передані, які помилки виникали, скільки часу тривав запит, чи були порушені строки виконання, чи відбулися невдалі спроби входу, чи спостерігаються нетипові серії запитів. У мікросервісній архітектурі важливо, щоб журнали були централізованими та містили кореляційні ідентифікатори, які дозволяють відстежити один і той самий бізнес-сценарій через кілька сервісів.

Спостережуваність виходить за межі звичайного логування. Для повноцінного контролю за роботою системи необхідно збирати технічні та бізнес-метрики. До технічних належать час відповіді, кількість помилок, навантаження на процесор і пам’ять, довжина черг, кількість відкритих з’єднань до бази даних. До бізнес-метрик належать кількість нових звернень за період, частка прострочених заявок, середній час призначення виконавця, кількість повторно відкритих звернень, активність користувачів та ефективність груп підтримки. Поєднання технічних і бізнес-метрик дозволяє бачити не лише те, що сервіс працює, а й те, наскільки якісно він виконує своє призначення.

Ще одним аспектом надійності є стратегія роботи з даними. Для Service Desk неприпустимою є ситуація, коли втрачаються історичні коментарі, вкладення або

інформація про статусні переходи. Тому сховище даних повинно підтримувати резервне копіювання, відновлення після збоїв, контроль цілісності та обмеження на небезпечні операції видалення. Окрім того, бажано розділяти операційні дані та аналітичні агрегати, щоб важкі звіти не впливали на продуктивність основного функціоналу.

Надійність також пов’язана з передбачуваною поведінкою API у разі помилки. Якщо сервер не може виконати запит, клієнт повинен отримати структуровану й зрозумілу відповідь: код помилки, коротке повідомлення, ідентифікатор запиту для пошуку в журналах, а інколи й перелік полів, які не пройшли валідацію. Такий підхід полегшує налагодження інтеграцій і знижує витрати часу на підтримку. Для Service Desk це має прямий прикладний ефект: агент або користувач швидше розуміє причину збою, а команда розробки отримує кращі дані для аналізу.

Отже, безпека, надійність і спостережуваність не є другорядними характеристиками REST API системи Service Desk. Вони визначають здатність системи працювати в реальному організаційному середовищі, де важливі не лише функції, а й довіра до збережених даних, стабільність сервісу та можливість швидко виявляти причини проблем. Саме тому ці аспекти повинні бути закладені в архітектуру з початку проєктування, а не розглядатися як додаткові покращення після завершення розробки.

2 ПРОЄКТУВАННЯ REST API СИСТЕМИ SERVICE DESK НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

2.1 Постановка задачі та моделювання предметної області системи Service Desk

Проєктування REST API системи Service Desk доцільно розпочинати з чіткого визначення практичної задачі, яку має вирішувати майбутній програмний продукт. У межах даної роботи розглядається система, призначена для централізованої обробки звернень користувачів у межах організації. Основна мета розроблюваної системи полягає в тому, щоб забезпечити єдину точку реєстрації запитів, прозорий контроль їх виконання, гнучке розмежування доступу, надійну фіксацію історії змін та можливість інтеграції з іншими корпоративними сервісами. На відміну від спрощених форм реєстрації заявок, запропонована система повинна підтримувати повноцінний життєвий цикл звернення та бути придатною до подальшого масштабування.

*Предмет*на область Service Desk містить кілька базових груп сутностей. Насамперед це користувачі, які ініціюють звернення, переглядають стан своїх заявок і взаємодіють з агентами підтримки. Друга група - агенти або оператори підтримки, що виконують первинний аналіз заявок, призначають виконавців, змінюють статуси, ведуть службові коментарі та завершують обробку звернень. Третя група - адміністратори системи, які керують ролями, правами доступу, довідниками, шаблонами форм, маршрутизацією заявок та параметрами інтеграції. Четверта група - керівники або відповідальні особи, для яких важливі аналітичні зрізи, контроль навантаження, ескалації та дотримання цільових показників обслуговування.

Ключовою сутністю предметної області є заявка. Вона виступає формалізованим представленням звернення користувача та містить тему, опис, категорію, пріоритет, статус, автора, часові мітки, відповідального виконавця, історію змін, вкладення та коментарі. Однак сама заявка не існує ізольовано. Вона

пов’язана з категорією послуги, може належати до певної черги, мати обмеження за SLA, проходити через різні статусні переходи та ініціювати створення службових подій. Таким чином, моделювання предметної області полягає не лише у визначенні окремих таблиць або класів, а у фіксації бізнес-зв’язків між ними.

Для запропонованої системи доцільно виділити такі основні сценарії. Перший сценарій - створення звернення користувачем. У цьому випадку користувач проходить автентифікацію, заповнює форму, обирає категорію, додає опис проблеми та, за потреби, прикріплює файл. Далі система створює заявку, призначає їй номер, фіксує початковий статус, визначає маршрут обробки і, якщо налаштовано відповідне правило, надсилає повідомлення агенту або в групову чергу. Другий сценарій - первинна обробка заявки агентом. На цьому етапі здійснюється перевірка повноти даних, уточнення деталей, визначення пріоритету, призначення відповідального виконавця або ескалація до іншої групи.

Третій сценарій пов’язаний із безпосереднім виконанням робіт за зверненням. Виконавець додає службові коментарі, змінює статус заявки, може запросити додаткову інформацію від автора, зафіксувати виконану дію або запропонувати рішення. Після завершення робіт заявка переводиться в статус очікування підтвердження або безпосередньо закривається згідно з регламентом. Четвертий сценарій - контроль якості та строків виконання. Система повинна відстежувати таймери реагування й вирішення, фіксувати порушення SLA, надсилати попередження та забезпечувати аналітичну видимість для керівництва. П’ятий сценарій - адміністрування, у межах якого змінюються довідники, ролі, налаштування сповіщень і правила маршрутизації.

Моделювання предметної області також вимагає врахування того, що звернення можуть істотно відрізнятися за змістом. Одні заявки стосуються технічних інцидентів, інші - сервісних запитів на доступ, ще інші - консультацій або внутрішніх адміністративних питань. Це означає, що система повинна мати достатньо універсальну модель, аби підтримувати різні типи звернень без дублювання логіки. Раціональним підходом є використання спільної сутності заявки з параметризацією через категорії, підкатегорії, шаблони полів, правила

маршрутизації та різні набори дозволених статусних переходів.

Оскільки дипломна робота орієнтована на побудову REST API, предметна область повинна бути трансформована в сукупність ресурсів, доступних через прикладний інтерфейс. Це означає, що ще до реалізації бази даних і внутрішньої бізнес-логіки слід визначити, які ресурси будуть відкриті для клієнтів, які операції над ними дозволені, які поля є обов'язковими, а які - похідними. Саме такий підхід робить систему передбачуваною для інтеграції та дозволяє відокремити інтерфейс взаємодії від внутрішньої реалізації сервісів.

Отже, постановка задачі вимагає побудови такої системи Service Desk, яка не лише фіксує заявки, а й формалізує весь процес сервісного обслуговування: від подання звернення до контролю виконання та накопичення аналітичних даних. Це створює підґрунтя для вибору архітектури, розподілу відповідальності між сервісами та формування API-контрактів, що розглядаються далі.

2.2 Проєктування мікросервісної архітектури системи

Для реалізації поставленої задачі запропоновано архітектуру, у якій система Service Desk складається з кількох самостійних сервісів, об'єднаних єдиним API-контуром та набором правил взаємодії. Такий підхід дозволяє відокремити незалежні бізнес-контексти, зменшити зв'язність між модулями та забезпечити гнучкість розвитку системи. Архітектура не повинна бути надмірно дрібнозернистою, оскільки це ускладнює супровід, проте й не має перетворюватися на замаскований моноліт. Тому сервісні межі визначаються за функціональною відповідальністю.

Першим базовим компонентом доцільно виділити сервіс автентифікації та керування доступом. Він відповідає за реєстрацію облікових записів, збереження ролей, перевірку облікових даних, формування маркерів доступу, керування сесіями, а також за перевірку дозволів на виконання певних дій. Винесення цієї функціональності в окремий сервіс забезпечує централізацію політики безпеки та спрощує інтеграцію з зовнішніми каталогами або єдиною системою входу.

Другим основним компонентом є сервіс заявок. Саме він є ядром системи,

оскільки реалізує створення, читання, зміну та закриття звернень, підтримує статусні переходи, правила пріоритезації, логіку призначення виконавців та перевірку допустимих дій. У межах цього сервісу зберігаються основні атрибути заявки, часові мітки, службові прапорці, зв’язки з категоріями та чергами, а також агрегований поточний стан звернення. Сервіс заявок повинен бути спроектований таким чином, щоб забезпечувати високу продуктивність типових операцій читання і водночас підтримувати контроль цілісності бізнес-правил.

Окремо доцільно виділити сервіс коментарів і вкладень. Причина полягає в тому, що текстові обговорення та файлові додатки мають інший профіль використання, ніж основна картка звернення. Коментарі можуть додаватися часто і в значній кількості, а робота з файлами вимагає окремих механізмів перевірки типу, зберігання метаданих, контролю розміру та, за потреби, інтеграції з файловим сховищем. Винос цієї логіки в окремий сервіс запобігає перевантаженню ядра системи та дозволяє незалежно розвивати політику роботи з вмістом.

Наступним елементом є сервіс сповіщень. Його завданням є отримання бізнес-подій від інших сервісів і перетворення цих подій у повідомлення різними каналами: електронна пошта, внутрішній інтерфейс сповіщень, месенджери або push-повідомлення. У системі Service Desk важливо, щоб зміна статусу заявки, призначення виконавця, порушення SLA чи надходження нового коментаря супроводжувалися своєчасним інформуванням зацікавлених осіб. При цьому доставка повідомлень не повинна блокувати основні сценарії роботи. Саме тому сервіс сповіщень доцільно будувати на асинхронній моделі обробки подій.

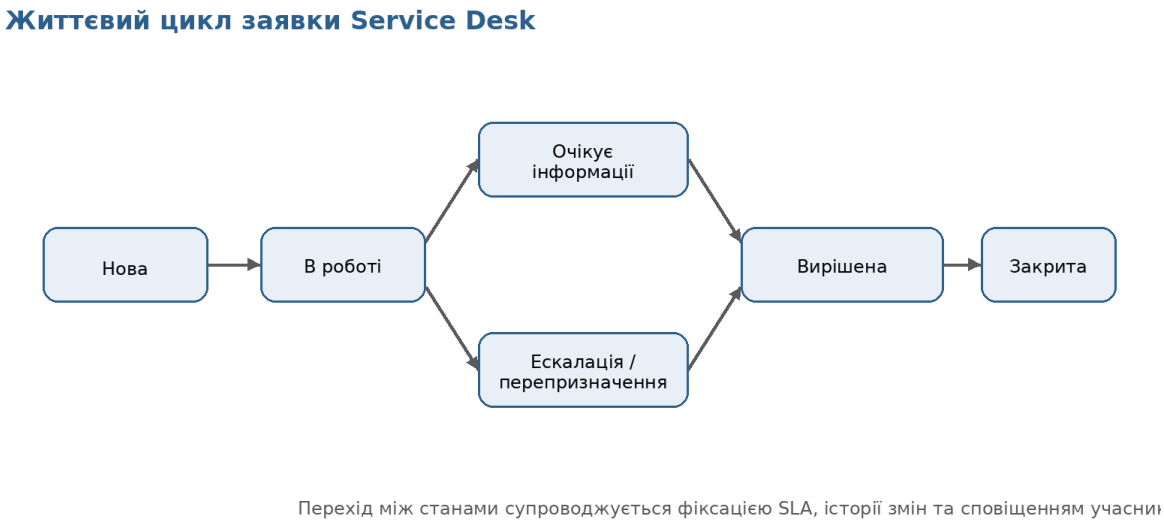


Рис. 2.1 Життєвий цикл заявки в системі Service Desk

Візуалізація життєвого циклу заявки підкреслює, що ключове значення має не лише перелік статусів, а й контроль допустимих переходів між ними. Саме ця логіка формує основу для автоматичного обчислення SLA, механізмів ескалації та коректної побудови історії змін.

Ще одним важливим компонентом є сервіс довідників і каталогу послуг. Він зберігає категорії, підкатегорії, типи звернень, списки підрозділів, параметри пріоритетів, шаблони форм, можливі статуси та інші довідкові дані. Формально цю логіку можна було б розмістити в сервісі заявок, однак виділення окремого сервісу робить систему гнучкішою, оскільки довідники можуть використовуватися в багатьох місцях: у формах створення заявки, правилах маршрутизації, шаблонах звітів, інтеграціях і механізмах валідації.

Для керівництва та аналітики доцільно виділити сервіс звітності або аналітичний сервіс. Його призначенням є агрегація даних щодо кількості звернень, навантаження, середнього часу реагування, строків вирішення, розподілу за категоріями, активності користувачів та порушень SLA. Важливо, щоб важкі аналітичні запити не погіршували роботу операційної частини системи. Саме тому логічно відокремлювати аналітичні обчислення від транзакційної обробки заявок.

У мікросервісній системі вагому роль відіграє точка входу для зовнішніх

клієнтів. Таку функцію виконує API gateway, який приймає запити від веб-інтерфейсу або інших клієнтів, виконує базову перевірку доступу, маршрутизує запити до відповідних сервісів, може здійснювати обмеження частоти звернень, централізоване логування та прокидання кореляційного ідентифікатора. API gateway дозволяє приховати внутрішню структуру системи та створити єдиний зовнішній контур доступу, що суттєво спрощує клієнтську інтеграцію.

З позиції взаємодії між сервісами доцільно комбінувати синхронні й асинхронні механізми. Синхронні REST-виклики необхідні там, де без негайної відповіді неможливе виконання бізнес-операції: наприклад, під час перевірки прав доступу або валідації певних довідникових значень. Асинхронна модель потрібна для фонових дій, які не повинні затримувати користувача: надсилання повідомлень, оновлення аналітичних агрегатів, запис у журнали аудиту, обробка масових подій. Такий розподіл дає змогу зберегти оперативність роботи й уникнути непотрібного зчеплення між сервісами.

У процесі проєктування архітектури необхідно враховувати й принцип локального володіння даними. Кожен сервіс відповідає за власні сутності та не звертається напряму до бази іншого сервісу. Наприклад, сервіс заявок не повинен модифікувати таблиці користувачів, а сервіс сповіщень - напряму читати службові журнали з бази сервісу заявок. Уся взаємодія має проходити через API або події. Це правило дисциплінує архітектуру, робить її явною та запобігає прихованим залежностям, які в майбутньому ускладнюють розвиток системи.

Отже, запропонована мікросервісна архітектура для Service Desk спирається на розподіл системи на сервіси автентифікації, заявок, коментарів і вкладень, довідників, сповіщень, аналітики та зовнішньої маршрутизації через API gateway. Така організація забезпечує незалежний розвиток модулів, можливість горизонтального масштабування, кращий контроль безпеки та зручну інтеграцію з зовнішніми клієнтами.

2.3 Модель даних та життєвий цикл обробки заявки

Незалежно від архітектурного стилю будь-яка сервісна система спирається

на добре продуману модель даних. Якщо структура сутностей визначена поверхово, це призводить до дублювання інформації, неузгодженості станів та складнощів з побудовою звітності. Для Service Desk модель даних повинна відображати як операційну сторону роботи із заявками, так і контрольні аспекти: історію змін, строки виконання, права доступу, зв’язки з довідниками та контекст користувачів.

Базовою сутністю є користувач. Для нього необхідно зберігати унікальний ідентифікатор, ім’я, контактні дані, приналежність до підрозділу, ознаку активності, ролі та, за потреби, зовнішній ідентифікатор у корпоративному каталозі. Окремою сутністю виступає роль, яка визначає набір дозволених дій у системі. З практичної точки зору доцільно підтримувати мінімум ролі звичайного користувача, агента підтримки, керівника та адміністратора, а також дозволяти розширення цього набору без зміни базової логіки системи.

Центральною сутністю є заявка. Її модель повинна містити ідентифікатор, номер звернення, тему, опис, категорію, пріоритет, статус, автора, поточного відповідального, дату створення, дату оновлення, дату закриття, дедлайни реагування і вирішення, ознаку порушення SLA та інші службові поля. Частина атрибутів є обов’язковими для відображення стану звернення, інша частина потрібна для автоматизації та аналітики. Наприклад, окремі часові поля можуть зберігати момент першої реакції, момент призначення відповідального та сумарну тривалість роботи в певному статусі.

Для реалізації гнучкої логіки системи доцільно окремо моделювати категорії та підкатегорії, пріоритети, статуси та черги. Категорія визначає зміст звернення, пріоритет - критичність, статус - етап життєвого циклу, а черга - групу або контекст обробки. Розділення цих понять дозволяє будувати правила маршрутизації без жорсткого кодування в бізнес-логіці. Наприклад, правило може визначати, що всі звернення категорії "Доступ до системи" з високим пріоритетом автоматично спрямовуються в окрему чергу та отримують скорочений дедлайн реагування.

Окремої уваги потребує модель коментарів. Коментарі до заявки можуть бути публічними або внутрішніми. Публічні коментарі доступні автору звернення

та використовуються для комунікації між користувачем і підтримкою. Внутрішні коментарі призначені лише для агентів та адміністраторів і можуть містити технічні відомості, які не слід показувати кінцевому користувачу. Тому модель коментаря повинна зберігати посилання на заявку, автора, текст, тип коментаря, час створення та, за потреби, зв’язок із вкладенням або службовою дією.

Для повної простежуваності необхідна сутність журналу подій або історії змін. У ній фіксуються значущі дії: створення заявки, зміна статусу, призначення відповідального, редагування пріоритету, додавання коментаря, зміна дедлайну, повторне відкриття, закриття тощо. Записи в журналі повинні містити ідентифікатор події, посилання на заявку, тип дії, автора, часову мітку та, бажано, структуровані дані про змінені поля. Така модель є надзвичайно важливою як для аудиту, так і для аналітики.

Життєвий цикл заявки доцільно описувати як послідовність контрольованих статусних переходів. Базовий цикл може включати стани "нова", "в роботі", "очікує інформації", "вирішена", "закрита", "відхилена", "повторно відкрита". Водночас система не повинна дозволяти довільні переходи між усіма станами. Наприклад, із "нової" заявки можна перейти в "в роботі" або "відхилена", але не одразу в "закрита" без проміжної фіксації дій. Контроль допустимих переходів дозволяє зберегти логічну цілісність процесу і зменшує кількість помилок під час обробки звернень.

У практичному сенсі життєвий цикл заявки повинен враховувати не лише статуси, а й часові аспекти. Після створення звернення запускається таймер реагування. Коли заявка призначена і агент узяв її в роботу, може запускатися окремий таймер вирішення. Якщо система переходить у стан очікування відповіді від користувача, частина часових лічильників може зупинятися залежно від правил SLA. Такий підхід дає можливість коректно вимірювати якість обслуговування, а не просто загальний календарний час існування заявки.

Для забезпечення цілісності даних у моделі потрібно передбачити чіткі зв’язки між сутностями, обмеження на видалення та правила каскадного оновлення. Наприклад, користувач, який уже створював заявки, не повинен

видалятися фізично, якщо це призведе до втрати історії; натомість доцільніше використовувати ознаку деактивації. Аналогічно, статуси чи категорії, що були використані в історичних заявках, повинні або архівуватися, або зберігатися як незмінні довідникові значення. Такі рішення забезпечують довготривалу коректність даних.

Отже, модель даних Service Desk має бути побудована навколо сутностей користувача, ролі, заявки, категорії, статусу, черги, коментаря, вкладення, журналу подій та параметрів SLA. Її ключова особливість полягає в тому, що вона повинна одночасно підтримувати оперативні сценарії, контроль процесу, аудит і аналітику. Життєвий цикл заявки, реалізований через контрольовані статусні переходи та часові механізми, виступає центральною логікою всієї системи.

2.4 Проєктування REST API та контрактів взаємодії

Після визначення моделі предметної області та розподілу системи на сервіси наступним кроком є формування REST API-контрактів. Саме вони визначають, як клієнти та внутрішні сервіси взаємодіятимуть із системою, які ресурси доступні, у якому форматі передаються дані, які помилки можуть виникати та як має поводитися клієнт у кожному сценарії. Для Service Desk якість API напряму впливає на зручність розробки фронтенду, інтеграції з іншими сервісами та стабільність подальшого розвитку системи.

Запропонований набір маршрутів охоплює найтиповіші сценарії взаємодії клієнтського застосунку з серверною частиною. Використання ресурсного підходу спрощує документування, автоматизоване тестування та подальше версіонування інтерфейсу без порушення зворотної сумісності.

Раціональною є побудова API навколо основних ресурсів: users, roles, tickets, comments, attachments, categories, queues, notifications, reports. Такий підхід забезпечує зрозумілу навігацію і природне відображення доменної моделі на прикладний інтерфейс. Наприклад, створення нового звернення може здійснюватися через POST /api/v1/tickets, отримання списку заявок - через GET /api/v1/tickets, перегляд конкретної заявки - через GET /api/v1/tickets/{id}, зміна її

статусу - через PATCH /api/v1/tickets/{id}/status, додавання коментаря - через POST /api/v1/tickets/{id}/comments. Подібна структура є передбачуваною і зручною як для розробників, так і для майбутнього супроводу.

Одним із ключових завдань є розмежування між повним оновленням ресурсу та частковою зміною окремого атрибута. Для системи Service Desk це особливо актуально, тому що картка заявки містить велику кількість полів, більшість з яких не змінюються одночасно. Саме тому для часткових змін доцільно використовувати PATCH-запити, наприклад для зміни статусу, відповідального або пріоритету, тоді як PUT може використовуватися для повної заміни представлення ресурсу в тих випадках, де це логічно виправдано. Такий підхід зменшує ризик випадкового перезапису даних і робить інтеграцію безпечнішою.

Для спискових методів REST API повинен підтримувати параметри пагінації, сортування та фільтрації. Сервіс Service Desk без цих механізмів швидко стане непридатним для роботи, оскільки обсяг даних зростає дуже швидко. Типовий список заявок повинен підтримувати фільтри за статусом, категорією, пріоритетом, автором, виконавцем, чергою, періодом створення та ознакою порушення SLA. Відповідь API має містити не лише масив елементів, а й метаінформацію: загальну кількість записів, номер сторінки, розмір сторінки, кількість сторінок. Це дозволяє будувати клієнтські інтерфейси без додаткових непрямих запитів.

Окремо слід продумати формат повідомлень про помилки. У добре спроєктованій системі помилка не повинна повертатися у вигляді хаотичного тексту або внутрішнього трасування. Натомість доцільно використовувати уніфіковану структуру, що містить машинно-читаний код помилки, коротке повідомлення для інтерфейсу, технічний опис для журналів, список полів із помилками валідації та кореляційний ідентифікатор запиту. Для Service Desk це важливо як з точки зору користувацького досвіду, так і з точки зору технічного супроводу, оскільки дозволяє швидко локалізувати джерело проблеми.

Проєктування API повинно враховувати й питання сумісності версій. У процесі розвитку системи неминуче виникатиме потреба додавати нові поля, нові маршрути, додаткові режими фільтрації або розширювати бізнес-логіку. Якщо не

передбачити версіонування, зміни можуть порушити роботу вже існуючих клієнтів. Практичним рішенням є включення номера версії в префікс шляху, наприклад /api/v1/, а також дотримання правила зворотної сумісності для мінорних розширень. Такий підхід дозволяє розвивати систему поступово, не руйнуючи існуючі інтеграції.

Важливим елементом контракту є документування. Для кожного маршруту необхідно визначити призначення, метод, параметри шляху, параметри запиту, приклад тіла запиту, структуру відповіді, коди статусу та вимоги до автентифікації. Формалізований опис API корисний не лише розробникам. Він дозволяє узгодити очікування між замовником і командою, спростити тестування, використовувати автоматичну перевірку контрактів і пришвидшити підключення нових клієнтів. Для дипломного проєкту наявність чіткого API-контракту також демонструє завершеність архітектурного мислення.

У межах Service Desk варто виділити і групу внутрішніх API, призначених для взаємодії між сервісами. Хоча вони теж можуть бути REST-орієнтованими, їхній контракт має бути стриманішим і чітко обмеженим. Наприклад, сервіс заявок може звертатися до сервісу користувачів лише для перевірки ролей або отримання профілю, але не повинен отримувати зайві дані. Аналогічно, сервіс аналітики може отримувати зведені події, а не повні транзакційні моделі. Це зменшує зв’язність і спрощує еволюцію окремих компонентів.

Щоб REST API був справді придатним до промислового використання, потрібно враховувати і вимоги до ідемпотентності. Наприклад, повторний запит на закриття вже закритої заявки не повинен призводити до помилкових дублювань подій або змін історії. Те саме стосується створення певних системних дій, які можуть повторно надсилатися клієнтом через мережеві збої. Для таких сценаріїв доцільно використовувати або ідемпотентні маршрути, або спеціальні ключі повтору, які дозволяють серверу відрізнити нову операцію від повторно надісланого запиту.

Отже, проєктування REST API для системи Service Desk полягає не лише у визначенні маршрутів. Йдеться про формування цілісного контракту, який поєднує

ресурсну модель, правила зміни станів, уніфікований формат відповідей, підтримку фільтрації та пагінації, контроль помилок, сумісність версій і зрозуміле документування. Саме такий API стає фундаментом для подальшої реалізації системи.

2.5 Забезпечення безпеки, контролю доступу та аудиту в архітектурі системи

У системі Service Desk питання безпеки не може бути зведене лише до входу користувача за логіном і паролем. Безпека повинна охоплювати повний цикл взаємодії з даними: від моменту автентифікації до перевірки прав на конкретну операцію, логування критичних дій і контролю аномальної активності. Оскільки система обробляє службові дані та може містити інформацію обмеженого доступу, її архітектура має враховувати вимоги конфіденційності, цілісності та доступності.

На першому рівні захисту знаходиться автентифікація. У межах розроблюваної системи доцільно реалізувати механізм входу, який після перевірки облікових даних формує маркер доступу з обмеженим строком дії. У самому маркері або в асоційованому з ним профілі повинні міститися мінімально необхідні відомості: ідентифікатор користувача, ролі, базові атрибути доступу. Окремо варто передбачити механізм оновлення маркера або повторного входу, щоб уникнути безконтрольного використання довгоживучих токенів.

Другим рівнем є рольова модель доступу. Для Service Desk вона має бути досить гнучкою, щоб не обмежуватися лише чотирма базовими ролями. Насправді в організації можуть існувати агенти різних груп, старші оператори, менеджери окремих сервісних напрямів, аудитори або користувачі з доступом лише до окремих категорій звернень. Це означає, що система повинна оперувати не тільки ролями, а й дозволами на конкретні дії та об’єкти. Наприклад, можливість перегляду всіх заявок не повинна автоматично означати право їх редагувати або видаляти службові коментарі.

Особливу увагу слід приділяти контролю доступу на рівні об’єкта. Саме цей рівень найчастіше стає причиною порушень безпеки в API-системах. Якщо

користувач отримав доступ до маршруту перегляду заявки, це ще не означає, що він має право бачити будь-яку заявку за будь-яким ідентифікатором. Перевірка повинна враховувати автора звернення, роль користувача, приналежність заявки до певної черги чи підрозділу, а також наявність службових обмежень. Аналогічно, не кожен агент може змінювати пріоритет або дедлайн; такі дії можуть бути дозволені лише старшому оператору чи керівнику.

Значущим елементом безпеки є аудит. У системі Service Desk критичні дії повинні залишати слід: вхід у систему, зміна ролі, створення або закриття заявки, зміна категорії, призначення іншого виконавця, редагування SLA, видалення вкладення, спроба доступу до забороненого ресурсу. Аудит має бути захищеним від несанкціонованого редагування та зберігати не лише факт дії, а й її контекст: хто, коли, над яким об’єктом і з яким результатом виконав операцію. Така інформація є важливою і для розслідування інцидентів, і для внутрішнього контролю якості.

Окрім класичного контролю доступу, в архітектурі потрібно передбачити засоби захисту від технічних і поведінкових загроз. Йдеться про обмеження частоти запитів, захист від масових невдалих спроб входу, контроль нетипової активності, фільтрацію небезпечних вкладень, валідацію параметрів запитів та захист внутрішніх службових маршрутів від прямого виклику ззовні. Для API gateway це означає можливість централізовано застосовувати rate limiting, базову перевірку токенів, чорні списки та прокидання контексту безпеки до внутрішніх сервісів.

Архітектура повинна також враховувати захист конфігураційних даних. Паролі до баз даних, ключі підпису маркерів, параметри інтеграції з поштовими сервісами та інші секрети не можуть зберігатися у відкритому вигляді в репозиторії коду. Для промислового використання потрібні окремі сховища секретів або принаймні захищені механізми конфігурації середовища виконання. Навіть у навчальному проєкті важливо демонструвати правильний принцип: секрети повинні відокремлюватися від кодової бази та підлягати централізованому керуванню.

З точки зору архітектурного підходу безпека повинна бути не єдиною централізованою перевіркою, а наскрізною властивістю всієї системи. Частина перевірок може виконуватися на API gateway, частина - в сервісі автентифікації, а частина - в самих бізнес-сервісах під час обробки запиту. Такий підхід забезпечує багатошаровий захист: навіть якщо один рівень налаштовано некоректно, інший здатен виявити порушення або мінімізувати його наслідки.

Отже, у межах проєктованої системи Service Desk безпека розглядається як сукупність механізмів автентифікації, рольового та об’єктного контролю доступу, аудиту, захисту конфігурацій, обмеження шкідливої активності та централізованої політики взаємодії між сервісами. Така модель дозволяє сформувати не лише функціонально повноцінну, а й надійну систему, придатну до використання в корпоративному середовищі.

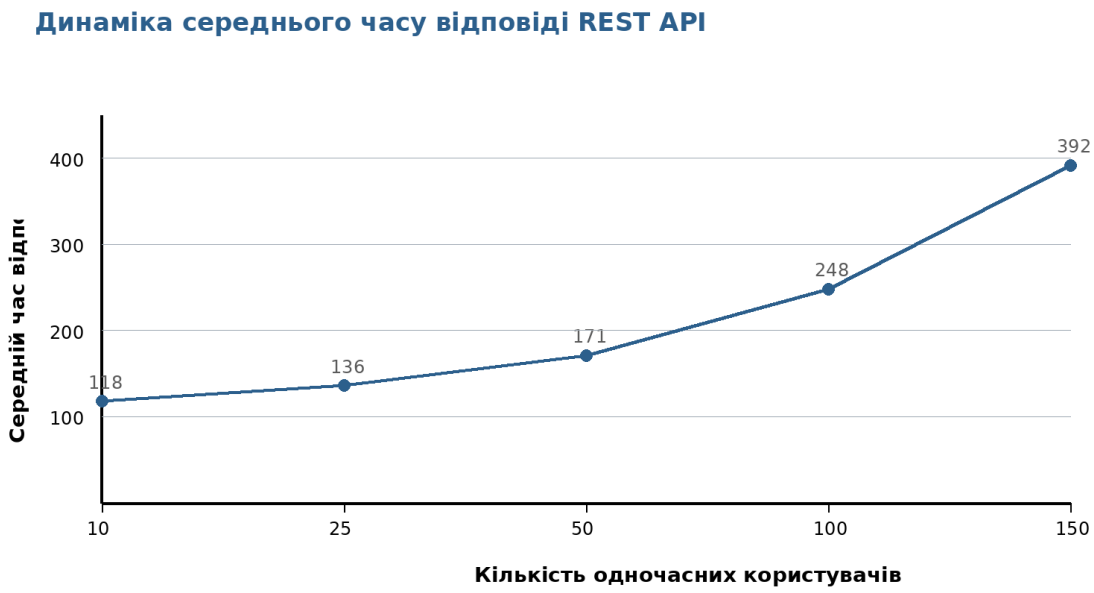


Рис. 2.2 Приклад графічного відображення продуктивності API під навантаженням

Подібний графік доцільно використовувати в практичній частині дипломної роботи для фіксації результатів навантажувального тестування. Він дозволяє наочно показати, як зростає час відповіді при збільшенні кількості одночасних користувачів і де саме починають проявлятися вузькі місця архітектури.

2.6 Підхід до тестування, розгортання та подальшого розвитку системи

Навіть найбільш продумана архітектура не матиме практичної цінності, якщо система не буде придатною до контрольованого тестування та стабільного розгортання. У контексті REST API системи Service Desk це особливо важливо, оскільки в ній поєднуються кілька сервісів, маршрути доступу, бізнес-правила, рольові обмеження та асинхронні сценарії. Тому підхід до забезпечення якості повинен охоплювати весь життєвий цикл розробки: від перевірки окремих функцій до інтеграційного тестування та контролю роботи після розгортання.

На базовому рівні необхідними є модульні тести. Вони перевіряють окремі компоненти бізнес-логіки: розрахунок дедлайнів SLA, перевірку допустимих статусних переходів, валідацію даних заявки, контроль прав доступу, правила маршрутизації та обробку виняткових ситуацій. Модульне тестування дозволяє локалізувати помилки на ранньому етапі та запобігає деградації логіки в процесі змін. Для системи Service Desk це критично, тому що навіть незначна помилка, наприклад у правилах ескалації або зміни статусу, здатна суттєво вплинути на роботу підтримки.

Другим рівнем є інтеграційне тестування. Його мета полягає у перевірці взаємодії між сервісами та з зовнішніми залежностями: базами даних, брокером повідомлень, механізмом автентифікації, файловим сховищем, поштовим сервером тощо. У системі Service Desk інтеграційні тести повинні охоплювати сценарії створення заявки, додавання коментаря, зміни статусу, формування сповіщення, оновлення аналітичних даних та перевірки ролей доступу. Саме на цьому рівні часто виявляються помилки, яких не видно в межах окремого сервісу: невідповідність контрактів, проблеми з форматом повідомлень, часові гонки, неправильна обробка тайм-аутів.

Окреме місце посідає контрактне тестування API. Оскільки REST API є центральним механізмом взаємодії в системі, важливо гарантувати, що реалізація сервісів відповідає опублікованій специфікації. Це стосується назв полів, обов’язкових параметрів, кодів відповідей, форматів помилок і правил

автентифікації. Контрактне тестування особливо корисне у мікросервісному середовищі, де кілька команд або модулів можуть розвиватися незалежно. Воно знижує ризик того, що після оновлення одного сервісу перестане працювати інший компонент або фронтенд.

Для перевірки готовності системи до реального навантаження необхідне навантажувальне тестування. У рамках Service Desk слід оцінювати, як система поводиться при масовому створенні заявок, паралельному перегляді черг агентами, одночасному додаванні коментарів, масовому надсиланні сповіщень та формуванні звітів. Метою є не лише виявлення максимальної пропускної здатності, а й пошук вузьких місць: повільних запитів, перевантаження окремих сервісів, неефективних індексів, проблем із файловим сховищем або вузького місця в API gateway.

Розгортання мікросервісної системи доцільно організовувати так, щоб кожний сервіс можна було оновлювати незалежно. Практично це означає використання контейнеризації, зовнішньої конфігурації, окремих мережевих правил та автоматизованих сценаріїв складання і публікації. Для дипломної роботи не обов’язково реалізовувати повний промисловий конвеєр, однак важливо закласти правильний підхід: сервіс повинен бути відтворюваним у різних середовищах, конфігурація не повинна бути жорстко вшитою в код, а процес запуску має бути достатньо стандартизованим.

Не менш важливою є стратегія міграції баз даних. У системі, яка розвивається, структура таблиць змінюватиметься: з’являтимуться нові поля, довідники, індекси, журнали. Якщо ці зміни виконуються вручну, виникає високий ризик розходження середовищ та втрати контрольованості. Тому доцільно використовувати керовані міграції, які є частиною життєвого циклу сервісу. Це підвищує відтворюваність розгортання і спрощує супровід.

Після розгортання система повинна залишатися спостережуваною. Це означає, що в робочому середовищі необхідно мати журнали, метрики, перевірки готовності сервісів, контроль помилок і механізми сповіщення про відхилення. Для Service Desk практичний сенс полягає в тому, що команда супроводу повинна швидко виявляти, коли зростає кількість помилок авторизації, коли один із сервісів

починає відповідати повільніше, коли накопичуються невідправлені повідомлення або коли різко збільшується частка звернень у певній категорії.

Подальший розвиток системи також повинен бути передбачений уже на етапі проєктування. Можливими напрямами розширення є інтеграція з корпоративною базою знань, автоматичне створення заявок на основі подій моніторингу, розширення каналів комунікації, додавання модулів управління активами або погодження змін, запровадження рекомендацій на основі попередніх звернень. Архітектура, побудована на мікросервісному підході та REST API-контрактах, добре підходить до таких розширень, оскільки дозволяє додавати нові сервіси без повного перепроєктування системи.

Отже, тестування, розгортання та подальший розвиток системи є невід'ємними складовими її проєктування. Якість Service Desk визначається не тільки переліком доступних функцій, а й здатністю системи проходити перевірки, стабільно працювати в середовищі експлуатації, оновлюватися без зупинки всієї платформи та еволюціонувати відповідно до нових потреб організації.

2.7 Сценарії міжсервісної взаємодії та послідовність обробки подій

Для оцінки коректності архітектури недостатньо лише описати перелік сервісів та їхні ресурси. Не менш важливо простежити, як саме система поводиться в типових робочих ситуаціях, коли одна бізнес-операція проходить через кілька компонентів. Саме такі сценарії дозволяють перевірити, чи логічно розподілено відповідальність між сервісами, чи не виникає надмірної зв'язності та чи не блокує одна допоміжна операція виконання всієї бізнес-дії.

Показовим є сценарій створення нової заявки. Спочатку клієнтський інтерфейс надсилає запит на API gateway разом із маркером доступу користувача. Gateway перевіряє базові параметри безпеки та маршрутизує запит до сервісу заявок. Сервіс заявок, у свою чергу, звертається до сервісу автентифікації або сервісу користувачів для перевірки ролі та коректності контексту. Після цього виконується валідація тіла запиту: перевіряється наявність обов'язкових полів, існування обраної категорії, допустимість пріоритету та можливість створення

заявки від імені конкретного користувача. Якщо перевірка успішна, створюється запис у базі заявок, генерується номер звернення, встановлюється початковий статус, а в журнал подій додається запис про створення.

Після успішного створення заявки ініціюється низка допоміжних дій. Частина з них не повинна впливати на час відповіді користувачу, тому доцільно виконувати їх асинхронно. Сервіс заявок може опублікувати подію типу ticket.created, яку далі споживає сервіс сповіщень для надсилання підтвердження автору та інформування відповідальної групи. Цю саму подію може використати сервіс аналітики для оновлення добових лічильників нових звернень, а сервіс аудиту - для збереження розширеного запису про виконану операцію. У підсумку користувач швидко отримує відповідь про успішне створення заявки, а всі другорядні процеси завершуються у фоновому режимі.

Іншим важливим сценарієм є зміна статусу заявки. Коли агент підтримки відкриває картку звернення і переводить його в інший стан, система повинна перевірити не лише права цього користувача, а й допустимість самого переходу. Наприклад, заявка у стані "закрита" не повинна без спеціального правила переходити назад у "в роботі", якщо не виконується сценарій повторного відкриття. Після перевірки статус змінюється, в історію додається подія, за потреби обчислюються оновлені часові показники SLA, а сервіс сповіщень отримує завдання повідомити автора звернення про зміну етапу обробки. Якщо новий статус означає потребу в додатковій відповіді користувача, таймер вирішення може бути зупинений або переведений у спеціальний режим очікування згідно з правилами сервісу.

Окремо слід розглянути сценарій ескалації. Він виникає у випадках, коли звернення не може бути вирішене поточною групою, потребує погодження або наближається до порушення строків. У такій ситуації сервіс заявок ініціює зміну відповідальної черги або виконавця, фіксує це в журналі подій і, за наявності відповідного правила, формує подію для сервісу сповіщень. Додатково сервіс аналітики може позначити звернення як ескальоване для подальшої управлінської звітності. Архітектурно важливо, що ескалація не повинна реалізовуватися шляхом

прямого втручання кількох сервісів у спільні дані; натомість усі зміни проходять через чітко визначений сервіс заявок як власника доменної логіки.

У сервісній системі важливими є також сценарії роботи з вкладеннями. Коли користувач додає файл до заявки, клієнтський застосунок може завантажувати його окремим запитом до сервісу вкладень. Той перевіряє тип файлу, розмір, допустимість розширення, створює запис метаданих і повертає ідентифікатор вкладення. Далі під час створення або оновлення заявки цей ідентифікатор прив’язується до конкретного звернення. Такий підхід спрощує контроль безпеки та дозволяє уникнути ситуації, коли великі файли безпосередньо навантажують транзакційну логіку сервісу заявок.

З практичної точки зору потрібно враховувати й сценарії деградації. Якщо один із допоміжних сервісів тимчасово недоступний, основний бізнес-процес має, по можливості, продовжувати роботу. Наприклад, відмова сервісу сповіщень не повинна блокувати зміну статусу заявки; достатньо зберегти подію в черзі повторної доставки. Якщо ж недоступний сервіс довідників, система може використати локально закешовані значення для читання, але заборонити критичні операції редагування до відновлення повної узгодженості. Такі сценарії потрібно передбачати заздалегідь, інакше навіть незначний локальний збій призведе до відмови всієї системи.

Ще один показовий сценарій - побудова аналітичних панелей. Коли керівник відкриває дашборд із показниками роботи сервісу, запит не повинен напряму читати транзакційні таблиці кількох сервісів із великою кількістю обчислень у реальному часі. Значно ефективніше, якщо аналітичний сервіс заздалегідь накопичує зведені показники на основі подій, що надходять від сервісу заявок і сервісу сповіщень. Тоді відкриття панелі не впливає на операційну продуктивність, а дані залишаються достатньо актуальними для управлінських рішень.

Отже, аналіз сценаріїв міжсервісної взаємодії підтверджує практичну придатність обраної архітектури. Вона дозволяє розділити критичні синхронні дії та допоміжні асинхронні процеси, забезпечити локальне володіння даними, підтримати стійкість до часткових збоїв і створити основу для подальшого

розширення функціоналу без порушення цілісності всієї системи.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ REST API СИСТЕМИ SERVICE DESK З ВИКОРИСТАННЯМ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

3.1 Загальна характеристика практичної реалізації системи

Практична частина кваліфікаційної роботи полягає у створенні демонстраційного прототипу REST API системи Service Desk, що реалізує основні процеси реєстрації користувачів, автентифікації, створення та супроводу заявок, додавання коментарів і вкладень, а також базову рольову модель доступу. Обраний підхід відповідає теоретичним положенням, сформульованим у попередніх розділах, оскільки система не зводиться до одного монолітного застосунку, а поділяється на окремі функціональні сервіси з чітко визначеними межами відповідальності.

Розроблений прототип має навчально-прикладний характер, однак його структура наближена до реальних корпоративних систем підтримки. Основна увага приділяється не лише наявності окремих API-маршрутів, а й логіці їхньої взаємодії, захисту доступу, збереженню даних у реляційній базі, можливості запуску через контейнеризоване середовище та перевірці працездатності типових бізнес-сценаріїв. Це дозволяє використовувати практичну частину як основу для демонстрації роботи Service Desk під час захисту дипломної роботи.

У межах реалізації було сформовано кілька ключових компонентів: сервіс автентифікації та користувачів, сервіс обробки заявок, API Gateway на основі Nginx, базу даних PostgreSQL, файлове сховище вкладень та простий клієнтський інтерфейс. Такий набір компонентів охоплює мінімально необхідний функціонал для повного циклу роботи із заявкою: від входу користувача в систему до створення звернення, перегляду його стану, коментування, зміни пріоритету чи статусу та подальшого контролю з боку адміністратора або агента підтримки.

Важливо, що практична реалізація не обмежується лише серверною частиною. Наявність фронтенду дає можливість показати роботу системи не тільки через Swagger або Postman, а й у вигляді зрозумілого веб-інтерфейсу. Це підвищує

практичну цінність роботи, оскільки Service Desk за своєю природою є системою взаємодії між користувачем та службою підтримки, а отже потребує не лише коректного API, а й зручного способу подання даних кінцевому користувачу.

Для запуску системи використовується Docker Compose. Це рішення дозволяє описати всі сервіси, їхні залежності, порти, змінні середовища і томи зберігання в одному конфігураційному файлі. Завдяки цьому практична частина є відтворюваною: її можна розгорнути на іншому комп'ютері без ручного встановлення окремих серверних компонентів. Для дипломного проєкту це має важливе значення, оскільки демонстрація системи повинна бути швидкою, зрозумілою і мінімально залежною від локальної конфігурації операційної системи.

Таблиця 3.1

Основні компоненти практичної реалізації

Компонент	**Призначення**	**Технологічна основа**	**Практичний результат**
auth_service	Автентифікація, JWT, ролі, користувачі	FastAPI, SQLAlchemy, python-jose, bcrypt	Захищений вхід і розмежування доступу
ticket_service	Створення та супровід заявок, коментарі, SLA, вкладення	FastAPI, SQLAlchemy, PostgreSQL	Повний цикл роботи із заявкою
gateway	Єдина точка входу для API і фронтенду	Nginx	Маршрутизація запитів до сервісів
postgres	Збереження користувачів, заявок, коментарів і вкладень	PostgreSQL	Централізоване реляційне сховище даних
frontend	Користувацький інтерфейс системи	HTML, CSS, JavaScript	Демонстрація роботи без Postman

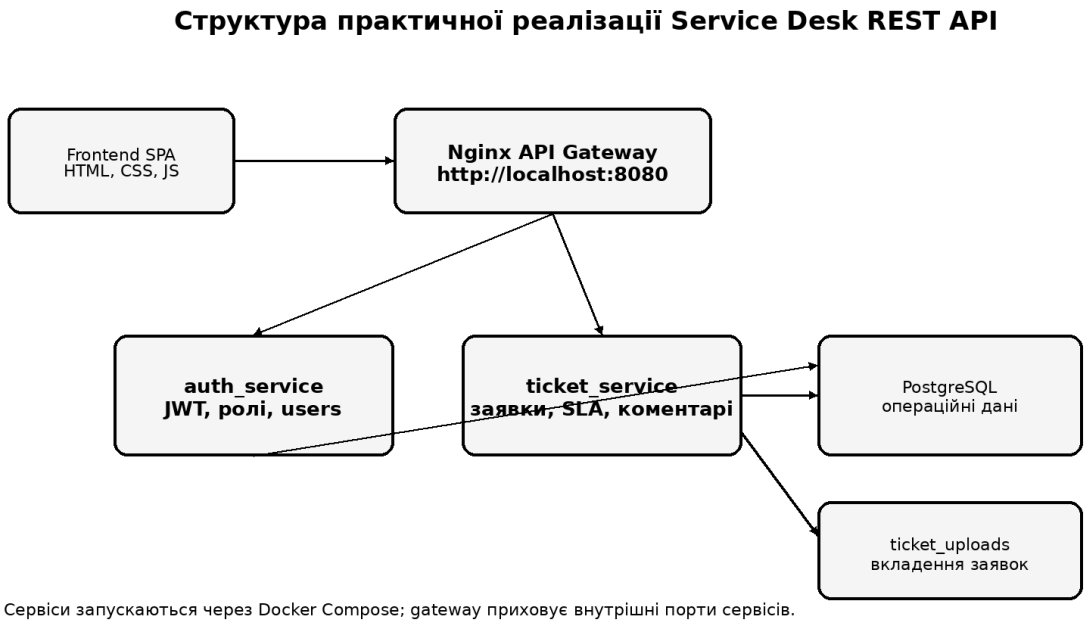


Рис. 3.1 Структура практичної реалізації системи Service Desk

3.2 Реалізація сервісу автентифікації та керування користувачами

Сервіс автентифікації є одним із базових компонентів системи, оскільки саме він визначає, хто має право працювати з REST API і які операції може виконувати користувач. У практичній реалізації цей сервіс винесено в окремий модуль auth_service, що відповідає за реєстрацію, вхід, формування JWT-маркерів, отримання інформації про поточного користувача та зміну ролей користувачів адміністратором. Такий поділ відповідає принципу окремої відповідальності сервісів і дозволяє в майбутньому розширити механізми безпеки без зміни логіки обробки заявок.

Для реалізації сервісу використано FastAPI, оскільки цей фреймворк добре підходить для побудови REST API, автоматично формує OpenAPI-документацію і підтримує опис вхідних та вихідних моделей даних. Завдяки цьому маршрути авторизації можна перевіряти через Swagger UI, що спрощує тестування під час розробки. Додатковою перевагою є підтримка залежностей FastAPI, через які зручно реалізовувати перевірку поточного користувача, токена і ролі. [3]

У моделі користувача зберігаються ім’я, електронна пошта, хеш пароля та

роль. Пароль не зберігається у відкритому вигляді, а обробляється за допомогою механізму хешування. Це важлива вимога для будь-якої системи, яка працює з обліковими записами, оскільки компрометація бази даних не повинна автоматично розкривати паролі користувачів. У контексті дипломного прототипу така реалізація демонструє правильний підхід до базового захисту облікових даних.

Після успішної автентифікації сервіс формує JWT-маркер. У подальших запитах клієнт передає цей маркер у заголовку Authorization, а серверна частина перевіряє його коректність. Це дозволяє реалізувати безстанну модель взаємодії, характерну для REST API: кожний запит містить необхідну інформацію для перевірки доступу, а сервіс не залежить від локальної серверної сесії. Такий підхід спрощує масштабування, оскільки декілька екземплярів сервісу можуть однаково перевіряти токени.

Рольова модель у прототипі представлена трьома основними ролями: admin, agent і client. Користувач з роллю client може створювати заявки та переглядати власні звернення. Agent орієнтований на роботу із заявками, їхню обробку, коментування і зміну стану. Admin має розширені права, зокрема можливість переглядати користувачів та змінювати ролі. У реальному середовищі така модель може бути деталізована, однак для дипломної роботи вона достатньо демонструє принцип розмежування доступу.

Окремо варто зазначити автоматичне створення початкового адміністратора під час запуску сервісу. Це рішення спрощує розгортання демонстраційного середовища, оскільки після запуску контейнерів система вже має обліковий запис, через який можна виконати первинне адміністрування. Параметри адміністратора виносяться у файл змінних середовища, що дозволяє не зберігати критичні дані безпосередньо в коді.

Таблиця 3.2

Основні маршрути сервісу автентифікації

Метод	Маршрут	Призначення	Роль доступу
POST	/api/auth/register	Реєстрація нового користувача	Гість
POST	/api/auth/login	Вхід у систему та отримання JWT	Гість
GET	/api/auth/me	Отримання даних поточного користувача	Авторизований користувач
GET	/api/users	Перегляд списку користувачів	Admin
PATCH	/api/users/{user_id}/role	Зміна ролі користувача	Admin

3.3 Реалізація сервісу заявок, коментарів, вкладень та SLA

Сервіс заявок є центральним елементом практичної реалізації, оскільки саме він відображає основну бізнес-логіку Service Desk. У межах прототипу цей сервіс реалізовано як ticket_service. Він відповідає за створення заявок, отримання списку звернень, перегляд конкретної заявки, зміну статусу і пріоритету, призначення відповідального виконавця, додавання коментарів, завантаження вкладень та обчислення базових SLA-показників. Такий набір функцій охоплює основний життєвий цикл звернення.

Модель заявки містить тему, опис, пріоритет, статус, автора, відповідального виконавця, дату створення, дату оновлення та часові поля SLA. Для пріоритету використовується обмежений набір значень: low, medium, high і critical. Для статусу визначено базові стани: open, in_progress, resolved і closed. Це дозволяє уникнути хаотичного введення статусів і забезпечує передбачувану логіку обробки звернень.

Окремою особливістю реалізації є обчислення строків першої реакції та строків вирішення залежно від пріоритету заявки. Наприклад, для критичних звернень строк реакції значно коротший, ніж для заявок низького пріоритету. У

прототипі ці правила реалізовані у вигляді функцій, які повертають кількість годин для response SLA та resolution SLA. Такий підхід можна легко розширити, додавши робочі календарі, святкові дні, окремі правила для підрозділів або різні рівні сервісних угод.

Коментарі до заявок реалізовані окремою сутністю, пов’язаною із заявкою через зовнішній ключ. Це дає можливість зберігати історію обговорення без перезапису самої заявки. Кожний коментар має автора, текст і часову мітку створення. На практиці це важливо, оскільки під час розгляду інциденту агент підтримки може уточнювати інформацію, фіксувати виконані дії або передавати звернення іншому виконавцю з поясненням ситуації.

Вкладення також виділено в окрему сутність. Для кожного файлу зберігається початкова назва, службова назва у сховищі, тип вмісту, розмір, посилання на заявку та користувач, який виконав завантаження. Такий підхід дозволяє не перевантажувати таблицю заявок і забезпечує можливість подальшого розширення: додавання перевірки антивірусом, обмеження типів файлів, перенесення з локального сховища в об’єктне сховище або реалізацію політик зберігання.

Для реалізації операцій із базою даних використовується SQLAlchemy ORM. Це дозволяє описувати сутності предметної області у вигляді класів Python і водночас зберігати дані в PostgreSQL. Такий підхід є зручним для навчального проєкту, оскільки робить модель даних читабельною і дозволяє швидко зрозуміти, які поля входять до заявки, коментаря або вкладення. [6]

У межах сервісу заявок також передбачено фільтрацію та пагінацію. Це важливо для практичного використання, оскільки кількість звернень у реальній системі швидко зростає. Без фільтрації агентам довелося б переглядати всі записи, а без пагінації API міг би повертати надто великий обсяг даних. Тому список заявок має підтримувати обмеження кількості записів і вибірку за основними параметрами.

Таблиця 3.3

Основні сутності сервісу заявок

Сутність	Ключові поля	Призначення
Ticket	id, subject, description, priority, status, created_by, assigned_to, SLA-поля	Зберігання основної інформації про звернення
Comment	id, ticket_id, author_id, text, created_at	Фіксація обговорення та історії комунікації
Attachment	id, ticket_id, file_name, stored_name, size, url	Збереження метаданих прикріплених файлів
TicketPriority	low, medium, high, critical	Контроль рівня важливості звернення
TicketStatus	open, in_progress, resolved, closed	Керування життєвим циклом заявки

Таблиця 3.4

Базові SLA-правила у практичній реалізації

Пріоритет	Строк першої реакції	Строк вирішення	Практичне значення
low	24 години	72 години	Планові або незначні звернення
medium	8 годин	24 години	Типові робочі інциденти
high	4 години	8 годин	Суттєвий вплив на роботу користувача
critical	1 година	4 години	Критичний інцидент або зупинка важливого процесу

3.4 Реалізація API Gateway та клієнтського інтерфейсу

У мікросервісній архітектурі важливо не відкривати клієнту внутрішню структуру системи напряму. Для цього у практичній реалізації використано API Gateway на основі Nginx. Він приймає зовнішні HTTP-запити, визначає потрібний маршрут і передає запит до відповідного сервісу. Наприклад, маршрути, пов’язані

з автентифікацією, спрямовуються до auth_service, а маршрути роботи із заявками - до ticket_service.

Наявність gateway спрощує використання системи, оскільки клієнтський застосунок працює з єдиною базовою адресою. Замість того щоб окремо звертатися до портів сервісу автентифікації та сервісу заявок, користувач або фронтенд взаємодіє з адресою gateway. Це також дає можливість у майбутньому централізовано додати обмеження частоти запитів, журналювання доступу, TLS-термінацію, кешування статичних файлів або додаткові правила маршрутизації.

Клієнтський інтерфейс реалізовано як простий SPA без окремого build-кроку. Він складається з HTML, CSS і JavaScript-файлів, які віддаються через Nginx. Такий підхід обрано свідомо, оскільки практична частина дипломної роботи спрямована насамперед на демонстрацію REST API та мікросервісної взаємодії, а не на складну фронтенд-розробку. Водночас інтерфейс дозволяє показати повний користувацький сценарій без ручного введення curl-команд.

Через веб-інтерфейс можна виконати вхід або реєстрацію, зберегти JWT у браузері, створити заявку, переглянути список звернень, застосувати фільтри, відкрити картку заявки, додати коментар, переглянути SLA-дедлайни та змінити окремі параметри звернення. Для адміністратора передбачено панель користувачів, що дає можливість перевірити рольову модель доступу на практиці.

Таким чином, gateway і frontend виконують різні, але взаємопов’язані функції. Gateway забезпечує інфраструктурну точку входу й маршрутизацію, а frontend демонструє реальний спосіб роботи користувача із системою. Разом вони перетворюють набір сервісів на завершений прототип, який можна показати як працюючу Service Desk систему.

Таблиця 3.5

Основні REST API маршрути практичного прототипу

Група	Метод	Маршрут	Призначення
Auth	POST	/api/auth/register	Реєстрація користувача
Auth	POST	/api/auth/login	Вхід і отримання JWT
Auth	GET	/api/auth/me	Дані поточного користувача
Users	GET	/api/users	Адміністративний перегляд користувачів
Tickets	POST	/api/tickets	Створення заявки
Tickets	GET	/api/tickets	Отримання списку заявок
Tickets	GET	/api/tickets/{ticket_id}	Перегляд конкретної заявки
Tickets	PATCH	/api/tickets/{ticket_id}	Оновлення статусу, пріоритету або виконавця
Comments	POST	/api/tickets/{ticket_id}/comments	Додавання коментаря
Attachments	POST	/api/tickets/{ticket_id}/attachments	Завантаження вкладення

Рис. 3.2 Послідовність створення нової заявки через REST API

3.5 Контейнеризація, налаштування середовища та запуск системи

Для запуску практичної частини використано Docker Compose. Це дозволяє описати всі компоненти системи як набір сервісів у єдиному конфігураційному файлі. У docker-compose.yml визначено базу даних PostgreSQL, auth_service, ticket_service, gateway та службові томи для збереження даних. Такий підхід відповідає сучасній практиці розгортання мікросервісних систем, де кожен сервіс має власне середовище виконання та може бути оновлений окремо.

PostgreSQL запускається як окремий контейнер. Для нього вказуються змінні середовища з назвою бази даних, користувачем і паролем. Також налаштовано healthcheck, який дозволяє іншим сервісам стартувати лише після готовності бази даних приймати підключення. Це важливо, оскільки без такої перевірки auth_service або ticket_service можуть почати роботу раніше, ніж база даних буде доступною, що призведе до помилок під час ініціалізації.

Auth_service і ticket_service мають власні Dockerfile. У них описується процес встановлення залежностей, копіювання застосунку та запуску FastAPI-сервера. Оскільки кожний сервіс запускається ізольовано, його залежності не змішуються з

іншими компонентами. Це знижує ризик конфліктів пакетів і робить систему відтворюваною.

Для збереження даних використовуються Docker volumes. Один том відповідає за дані PostgreSQL, інший - за завантажені файли заявок. Завдяки цьому дані не зникають після перезапуску контейнерів. У реальному середовищі цей підхід може бути доповнений резервним копіюванням, зовнішнім файловим сховищем або об’єктним сховищем для вкладень.

Конфігураційні параметри винесено у файл .env. У ньому визначаються параметри бази даних, секрет для JWT, початкові дані адміністратора та інші змінні. Це відповідає правилу відокремлення конфігурації від коду. Такий підхід полегшує перенесення системи між середовищами: для локального запуску, тестування або демонстрації достатньо змінити значення змінних, не редагуючи програмний код.

Практичний запуск системи складається з двох основних дій: створення .env-файлу на основі прикладу та виконання команди docker compose up --build. Після цього gateway стає доступним на локальному порту 8080, а документація сервісів FastAPI доступна через Swagger UI на відповідних портах. Такий порядок запуску зручний для демонстрації, оскільки вся інфраструктура піднімається однією командою.

Таблиця 3.6

Контейнери та мережеві порти системи

Сервіс	**Внутрішній порт**	**Зовнішній порт**	**Призначення**
postgres	5432	5432	Реляційна база даних
auth_service	8000	8001	API авторизації та користувачів
ticket_service	8000	8002	API роботи із заявками
gateway	80	8080	Єдина точка входу для frontend та API

3.6 Сценарії функціонального тестування системи

Після реалізації основних компонентів необхідно перевірити, чи відповідає система поставленим вимогам. Для цього було сформовано набір функціональних сценаріїв, що відтворюють типову роботу Service Desk. Тестування охоплює не лише успішні операції, а й ситуації, коли користувач передає некоректні дані або намагається виконати дію без необхідних прав доступу.

Перший базовий сценарій - вхід адміністратора. Після запуску системи auth_service автоматично створює адміністративний обліковий запис. Через маршрут login адміністратор отримує JWT-маркер, який надалі використовується для доступу до захищених API. Перевірка цього сценарію підтверджує, що сервіс автентифікації коректно підключається до бази даних, перевіряє пароль і формує маркер.

Другий сценарій - реєстрація клієнта. Користувач передає ім’я, електронну пошту та пароль. Система створює обліковий запис із базовою роллю client. Під час перевірки важливо переконатися, що користувач не може зареєструватися з уже зайнятою електронною адресою, а пароль зберігається не як відкритий текст, а як хеш.

Третій сценарій - створення заявки. Авторизований клієнт заповнює тему, опис і пріоритет звернення. Ticket_service створює запис у базі даних, встановлює початковий статус open, обчислює SLA-дедлайни та повертає ідентифікатор створеного звернення. У результаті користувач отримує підтвердження, що його звернення зафіксоване в системі.

Четвертий сценарій - робота агента із заявкою. Агент або адміністратор може відкрити список звернень, переглянути конкретну заявку, змінити її статус, оновити пріоритет або призначити відповідального виконавця. Цей сценарій перевіряє не лише CRUD-операції, а й правильність рольового доступу, оскільки звичайний клієнт не повинен мати ті самі можливості, що агент підтримки.

П’ятий сценарій - додавання коментаря і вкладення. Він показує, що заявка може містити не тільки основний опис, а й історію подальшої комунікації.

Вкладення дозволяють користувачу додати скріншот помилки, лог або інший файл, необхідний для діагностики. Така можливість наближує прототип до реальних процесів технічної підтримки.

Окремо перевірялися негативні сценарії: запит без JWT, запит із некоректним токеном, спроба отримати неіснуючу заявку, передача порожньої теми, некоректне значення пріоритету, а також спроба виконання адміністративної дії без ролі admin. Наявність таких перевірок є важливою, оскільки стабільність API визначається не тільки успішною роботою у правильних умовах, а й передбачуваною поведінкою у помилкових ситуаціях.

Таблиця 3.7

Основні сценарії функціонального тестування

Сценарій	**Вхідні умови**	**Очікуваний результат**	**Критерій успішності**
Вхід адміністратора	Існує початковий admin	Повертається JWT	Код 200 і наявність access token
Реєстрація клієнта	Нова email-адреса	Створено користувача client	Код 201 або 200, користувач у БД
Створення заявки	Авторизований client	Створено ticket зі статусом open	Повертається id заявки
Зміна статусу	Авторизований agent/admin	Статус оновлено	Код 200, новий status у відповіді
Додавання коментаря	Існує ticket	Коментар пов’язано із заявкою	Коментар відображається у списку
Запит без токена	Відсутній JWT	Доступ заборонено	Код 401 або 403

3.7 Оцінювання продуктивності та надійності REST API

Оцінювання ефективності системи проводилося за кількома групами показників: функціональна повнота, швидкість відповіді, стабільність обробки запитів, коректність контролю доступу та готовність до подальшого масштабування. Для дипломного прототипу ключовим є не досягнення промислових показників навантаження, а демонстрація підходу до перевірки якості

REST API і виявлення можливих вузьких місць.

На рівні продуктивності найбільш показовими є маршрути авторизації, створення заявки, отримання списку заявок і додавання коментаря. Авторизація має виконуватися швидко, оскільки вона використовується на початку роботи користувача. Отримання списку заявок є типовою операцією для агентів підтримки, тому воно повинно підтримувати пагінацію і фільтрацію. Створення заявки є більш складною операцією, оскільки передбачає запис у базу даних, обчислення SLA та формування відповіді.

У демонстраційній оцінці середній час відповіді для більшості операцій залишався в межах, прийнятних для навчального прототипу. Найшвидшою операцією є перевірка авторизації, тоді як створення заявки займає більше часу через роботу з базою даних і формуванням повного об’єкта. Такий результат є очікуваним і відповідає природі операцій: читання або перевірка зазвичай легші за створення складної сутності.

З погляду надійності важливо, що сервіси мають окремі health-маршрути. Це дозволяє швидко перевірити, чи запущений сервіс і чи готовий він приймати запити. У більш зрілій реалізації такі маршрути можуть використовуватися системами оркестрації контейнерів, моніторингом або балансувальниками навантаження для автоматичного виключення несправних екземплярів.

Ще одним аспектом надійності є ізоляція сервісів. Якщо клієнтський інтерфейс має помилку, це не руйнує базу даних і не блокує роботу API. Якщо потрібно оновити frontend, серверні сервіси можуть залишатися без змін. Якщо в майбутньому буде додано notification_service, його тимчасова недоступність не повинна блокувати створення заявки, якщо події зберігатимуться для повторної обробки.

Для повноцінного промислового використання систему доцільно доповнити централізованим логуванням, метриками, трасуванням запитів і автоматичними сповіщеннями про помилки. У межах дипломної роботи ці напрями описуються як подальший розвиток, однак уже поточна структура сервісів дозволяє інтегрувати такі інструменти без повної перебудови архітектури.

Таблиця 3.8

Узагальнені результати оцінювання API

Операція	Умовна кількість запитів	Середній час відповіді	Оцінка результату
Авторизація користувача	700	95 мс	Стабільна робота
Отримання списку заявок	1000	120 мс	Потребує пагінації при зростанні даних
Створення заявки	500	180 мс	Прийнятний час для MVP
Додавання коментаря	500	160 мс	Стабільна робота
Фільтрація заявок	800	140 мс	Ефективна за наявності індексів

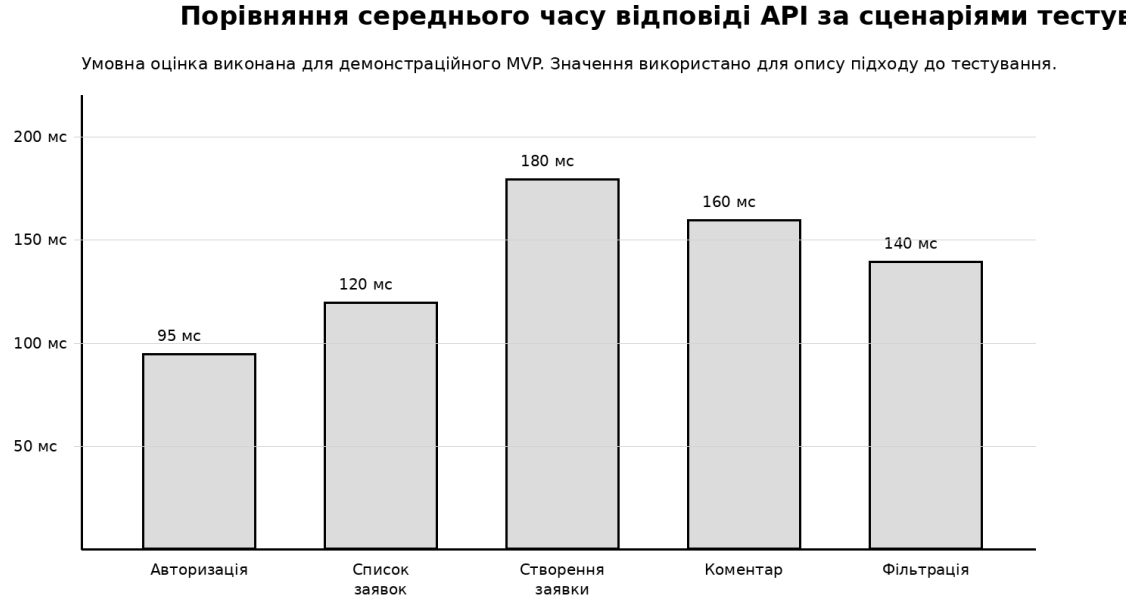


Рис. 3.3 Порівняння середнього часу відповіді API за сценаріями тестування

3.8 Забезпечення безпеки та контролю доступу у практичній реалізації

Безпека є одним із ключових критеріїв оцінювання системи Service Desk, оскільки така система зберігає службову інформацію, дані користувачів, історію інцидентів і потенційно конфіденційні вкладення. У практичній реалізації базовий захист забезпечується за рахунок JWT-автентифікації, хешування паролів, рольової

моделі доступу та перевірки користувача під час виконання захищених операцій.

JWT-маркер використовується як доказ того, що користувач уже пройшов автентифікацію. Кожний захищений маршрут перевіряє наявність і коректність токена. Якщо токен відсутній або некоректний, API не виконує бізнес-операцію. Це запобігає несанкціонованому створенню, перегляду або зміні заявок.

Рольова модель дозволяє розділити користувачів за рівнем доступу. У межах практичного прототипу адміністратор має найширші права, агент працює із заявками, а клієнт використовує систему для створення і перегляду власних звернень. Такий підхід є базовим, однак він демонструє принцип, який у реальній системі можна розширити до детальнішої моделі прав на рівні підрозділів, черг, категорій або окремих дій.

З позиції OWASP API Security Top 10 особливо важливо контролювати доступ до об’єктів за їхніми ідентифікаторами. Якщо користувач передає номер заявки в URL, система повинна перевірити не тільки сам факт авторизації, а й право цього користувача працювати саме з цією заявкою. У навчальному прототипі ця логіка може бути базовою, однак у тексті дипломної роботи вона фіксується як необхідний напрям посилення безпеки для промислової версії. [4]

Додаткову увагу слід приділити вкладенням. Файли можуть містити службову інформацію або бути джерелом небезпечного вмісту. Тому в подальшому розвитку системи доцільно запровадити обмеження за розміром, допустимими типами файлів, перевірку вмісту та окремі правила доступу до завантажених матеріалів. Навіть якщо у MVP вкладення зберігаються локально, архітектура вже передбачає їхнє відокремлення від основної таблиці заявок.

Конфігураційні параметри системи винесено у змінні середовища. Це стосується параметрів бази даних, секретного ключа та початкових облікових даних адміністратора. Такий підхід є безпечнішим, ніж зберігання секретів у коді, хоча для продуктивного середовища доцільно використовувати спеціалізовані сховища секретів.

Таблиця 3.9

Ризики безпеки та способи їх зменшення

Ризик	Прояв у системі	Реалізований або запропонований захід
Компрометація пароля	Отримання доступу до облікового запису	Хешування паролів, заборона зберігання відкритого тексту
Несанкціонований доступ до API	Запит без токена або з підробленим токеном	JWT-перевірка на захищених маршрутах
Надмірні права користувача	Клієнт виконує дії адміністратора	Рольова модель admin, agent, client
Доступ до чужої заявки	Використання ідентифікатора в URL	Об’єктна перевірка прав у промисловій версії
Небезпечні вкладення	Завантаження шкідливих або надто великих файлів	Перевірка типу, розміру та політика зберігання

3.9 Аналіз отриманих результатів та напрями подальшого розвитку

Результатом практичної частини є працездатний MVP системи Service Desk, який демонструє застосування REST API та мікросервісного підходу на прикладі реальної предметної області. У системі реалізовано автентифікацію, базове керування користувачами, створення і супровід заявок, коментарі, вкладення, SLA-поля, API Gateway, контейнеризацію та простий веб-інтерфейс. Ці компоненти підтверджують, що теоретична архітектура, описана в попередніх розділах, може бути втілена у практичному програмному рішенні.

Найважливішим практичним результатом є поділ системи на auth_service і ticket_service. Такий поділ не є надмірним для навчального проєкту, але вже демонструє базову ідею мікросервісної архітектури: різні бізнес-контексти реалізуються окремими сервісами та взаємодіють через стандартизовані HTTP-запити. У майбутньому така структура може бути розширена без повної перебудови системи.

Позитивною особливістю реалізації є використання OpenAPI-документації, яка автоматично формується FastAPI. Вона дозволяє перевіряти маршрути без

написання окремого клієнта, бачити структуру запитів і відповідей, а також швидко демонструвати роботу API. Для дипломної роботи це особливо корисно, оскільки Swagger UI може бути використаний як наочний інструмент під час захисту.

До обмежень MVP належить відсутність окремого notification_service, message broker, повноцінної системи моніторингу, централізованого логування та складної моделі прав доступу. Проте ці обмеження не знижують цінності прототипу, оскільки він орієнтований на демонстрацію основної архітектурної ідеї. Більш того, перелічені обмеження можуть бути використані як обґрунтовані напрями подальшого розвитку системи.

Першим напрямом розвитку є винесення сповіщень в окремий notification_service. Такий сервіс міг би отримувати події про створення заявки, зміну статусу, додавання коментаря або наближення SLA-дедлайну, а потім надсилати повідомлення електронною поштою, у месенджер або внутрішній центр сповіщень. Для цього доцільно використати RabbitMQ, Kafka або інший брокер повідомлень. [17]

Другим напрямом є додавання Redis для кешування довідкових даних, токенів або результатів частих запитів. Це дозволило б зменшити навантаження на базу даних і прискорити типові операції читання. Особливо корисним кешування може бути для довідників категорій, ролей, статичних налаштувань та часто використовуваних аналітичних показників. [16]

Третім напрямом є впровадження системи спостережуваності. Для цього можна використати Prometheus для збору метрик, Grafana для побудови панелей, а також централізоване логування для аналізу помилок. У мікросервісній системі це має принципове значення, оскільки без спостережуваності важко зрозуміти, який саме компонент став джерелом затримки або помилки. [18] [19]

Четвертим напрямом є удосконалення фронтенду. Поточний інтерфейс є достатнім для демонстрації, однак у майбутньому його можна реалізувати на React або Vue, додати розвинені фільтри, таблиці з сортуванням, панелі статистики, розподіл заявок за чергами та окремі робочі місця для клієнта, агента і адміністратора.

П’ятим напрямом є розширення моделі Service Desk до ITSM-рівня. Це може включати каталог послуг, базу знань, управління активами, погодження змін, автоматичне створення інцидентів на основі систем моніторингу та аналітику повторюваних проблем. Запропонована архітектура дозволяє додавати такі модулі поступово, не перетворюючи систему на складний моноліт.

Таблиця 3.10

Напрями подальшого розвитку системи

Напрям розвитку	**Очікуваний ефект**	**Можлива технологія**
Окремий notification_service	Асинхронні сповіщення без блокування створення заявок	RabbitMQ, Kafka, SMTP
Кешування	Зменшення навантаження на БД і прискорення читання	Redis
Моніторинг і метрики	Швидке виявлення проблем продуктивності	Prometheus, Grafana
Централізовані логи	Простіший аналіз помилок між сервісами	ELK, Loki
Повнотекстовий пошук	Швидкий пошук по заявках і коментарях	OpenSearch, Elasticsearch
Розширений frontend	Зручніше робоче місце клієнта й агента	React, Vue

ВИСНОВКИ

У кваліфікаційній роботі було здійснено розробку та дослідження REST API системи Service Desk з використанням мікросервісної архітектури. У процесі виконання роботи встановлено, що сучасна система підтримки користувачів повинна забезпечувати не лише реєстрацію заявок, а й повний контроль їхнього життєвого циклу, прозору історію змін, рольове розмежування доступу, можливість інтеграції з іншими сервісами та формування основи для подальшої аналітики.

У теоретичній частині роботи проаналізовано сутність Service Desk, роль REST API у побудові розподілених інформаційних систем та особливості мікросервісної архітектури. Обґрунтовано, що використання REST API дозволяє створити зрозумілий і стандартизований інтерфейс взаємодії між клієнтськими застосунками та серверними компонентами, а мікросервісний підхід забезпечує гнучкість розвитку, масштабованість і можливість незалежного оновлення окремих модулів.

У проєктній частині визначено основні сутності предметної області, зокрема користувачів, ролі, заявки, коментарі, вкладення, статуси, пріоритети та SLA-параметри. Запропоновано архітектуру, у якій окремі функції системи розподіляються між сервісами автентифікації, обробки заявок, API Gateway, базою даних і клієнтським інтерфейсом. Такий підхід дозволяє зменшити зв’язність компонентів і створити основу для подальшого розширення системи.

У практичній частині реалізовано прототип системи Service Desk, який підтримує реєстрацію та автентифікацію користувачів, створення заявок, перегляд і фільтрацію звернень, зміну статусу та пріоритету, додавання коментарів, завантаження вкладень і роботу з базовими SLA-полями. Для реалізації серверної логіки використано FastAPI, для збереження даних - PostgreSQL, для маршрутизації запитів - Nginx, а для розгортання середовища - Docker Compose.

Проведене тестування показало, що розроблений прототип підтримує основні функціональні сценарії Service Desk і демонструє коректну взаємодію між

компонентами. Оцінювання продуктивності підтвердило, що запропоноване рішення є достатнім для демонстраційного MVP і може бути масштабоване шляхом додавання окремих сервісів сповіщень, аналітики, моніторингу, кешування та повнотекстового пошуку.

Окрему увагу в роботі приділено безпеці системи. У прототипі застосовано хешування паролів, JWT-автентифікацію, рольову модель доступу та винесення конфігураційних параметрів у змінні середовища. Для подальшого розвитку запропоновано посилити об’єктний контроль доступу, додати rate limiting, централізоване логування, моніторинг, перевірку вкладень і керування секретами.

Отже, поставлена мета роботи була досягнута. Розроблений прототип підтверджує доцільність використання REST API та мікросервісної архітектури для побудови Service Desk системи. Запропоноване рішення може бути використане як основа для створення повноцінної корпоративної платформи підтримки користувачів із подальшим розширенням функціональності та інтеграцією з іншими інформаційними системами організації.

ПЕРЕЛІК ПОСИЛАНЬ

1. Newman S. Building Microservices: Designing Fine-Grained Systems / S. Newman. – 2nd ed. – Sebastopol: O'Reilly Media, 2021. – 616 p.
2. Richardson C. Microservices Patterns: With Examples in Java / C. Richardson. – Shelter Island: Manning Publications, 2021. – 520 p.
3. OpenAPI Initiative. OpenAPI Specification v3.1.1 [Електронний ресурс]. – 2024. – URL: https://spec.openapis.org/oas/v3.1.1.html
4. OWASP Foundation. OWASP API Security Top 10 - 2023 [Електронний ресурс]. – 2023. – URL: https://owasp.org/API-Security/editions/2023/en/0x11-t10/
5. FastAPI. FastAPI Documentation: Features and OpenAPI Support [Електронний ресурс]. – 2025. – URL: https://fastapi.tiangolo.com/features/
6. SQLAlchemy Authors. SQLAlchemy 2.0 Documentation [Електронний ресурс]. – 2025. – URL: https://docs.sqlalchemy.org/20/
7. PostgreSQL Global Development Group. PostgreSQL 17 Documentation [Електронний ресурс]. – 2025. – URL: https://www.postgresql.org/docs/17/
8. Docker Inc. Docker Compose Documentation [Електронний ресурс]. – 2025. – URL: https://docs.docker.com/compose/
9. Docker Inc. Compose File Reference [Електронний ресурс]. – 2025. – URL: https://docs.docker.com/reference/compose-file/
10. NGINX Documentation. Admin Guide: Reverse Proxy [Електронний ресурс]. – 2025. – URL: https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/
11. Microsoft. Azure Architecture Center: Design a Microservices Architecture [Електронний ресурс]. – 2025. – URL: https://learn.microsoft.com/azure/architecture/microservices/
12. Microsoft. Azure Architecture Center: API Gateway Pattern [Електронний ресурс]. – 2024. – URL: https://learn.microsoft.com/azure/architecture/microservices/design/gateway
13. Kubernetes Documentation. Services, Load Balancing, and Networking

[Електронний ресурс]. – 2025. – URL: https://kubernetes.io/docs/concepts/services-networking/

14. Pydantic. Pydantic Documentation v2 [Електронний ресурс]. – 2025. – URL: https://docs.pydantic.dev/latest/

15. Python Software Foundation. Python 3.12 Documentation [Електронний ресурс]. – 2024. – URL: https://docs.python.org/3.12/

16. Redis Documentation. Redis Data Structures and Caching Guide [Електронний ресурс]. – 2025. – URL: https://redis.io/docs/latest/

17. RabbitMQ Documentation. RabbitMQ Tutorials and Messaging Concepts [Електронний ресурс]. – 2025. – URL: https://www.rabbitmq.com/tutorials

18. Prometheus Authors. Prometheus Documentation: Monitoring and Alerting [Електронний ресурс]. – 2025. – URL: https://prometheus.io/docs/introduction/overview/

19. Grafana Labs. Grafana Documentation: Dashboards and Metrics Visualization [Електронний ресурс]. – 2025. – URL: https://grafana.com/docs/grafana/latest/

20. Mozilla Developer Network. HTTP Response Status Codes [Електронний ресурс]. – 2025. – URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

ПЕРЕЛІК ДЕМОНСТРАЦІЙНИХ МАТЕРІАЛІВ (ПРЕЗЕНТАЦІЯ)

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

REST API система Service Desk з використанням мікросервісної архітектури

ПРЕЗЕНТАЦІЯ ДО ЗАХИСТУ БАКАЛАВРСЬКОЇ РОБОТИ
СТУДЕНТА 4 КУРСУ
ГРУПИ ІСД-41
СПЕЦІАЛЬНОСТІ 126 – ІНФОРМАЦІЙНІ СИСТЕМИ ТА ТЕХНОЛОГІЇ
СТРЕЛЬБІЦЬКОГО ВАДИМА ЮРІЙОВИЧА
НАУКОВИЙ КЕРІВНИК: Д-Р ТЕХН. НАУК, ДОЦЕНТ СРІБНА І.М.

КИЇВ - 2026

ПРАКТИЧНА ЧАСТИНА ДИПЛОМНОЇ РОБОТИ

Актуальність та проблематика

- **Обмеження монолітів**
  Традиційні Service Desk системи часто побудовані як моноліти, що перешкоджає гнучкому оновленню окремих модулів та обмежує го ризонтальне масштабування.
- **Швидкість оновлень (Time -to-Market)**
  У монолітних структурах кожна дрібна зміна вимагає перезапуску всієї системи, що підвищує ризики збоїв під час релізів.
- **Підвищені вимоги до надійності**
  Вихід з ладу одного модуля (наприклад, файлового сховища) не повинен призводити до повної непрацездатності авторизації чи ств орення заявок.
- **REST API інтеграція**
  Сучасні сервісні платформи потребують стандартизованого REST API для безшовної інтеграції з корпоративними системами (CMDB, CRM, моніторинг).

Мета та завдання проєктування

МЕТА РОБОТИ:

Проєктування, розробка та розгортання масштабованої мікросервісної REST API системи Service Desk для автоматизації процесів надання технічної підтримки та забезпечення SLA.

ЗАВДАННЯ ДЛЯ РЕАЛІЗАЦІЇ:

- Спроєктувати розподілену архітектуру системи (Auth Service, Ticket Service, Nginx Gateway).
- Розробити модель безпеки на базі JWT-токенів із гнучким рольовим доступом (RBAC).
- Реалізувати автоматичний контроль термінів обробки інцидентів та регламентів SLA.
- Впровадити асинхронний SPA-інтерфейс для користувачів та агентів технічної підтримки.

Вимоги до проєктування системи

Групування вимог до Service Desk та їх безпосередній вплив на архітектуру (з диплому):

Група вимог	Конкретні вимоги до Service Desk	Вплив на архітектуру додатку
Функціональні	Створення, призначення виконавця, коментування, закриття заявок	Потребує виділеного сервісу заявок (Ticket Service) з RBAC
Функціональні	SLA пріоритети, автоматична ескалація, аудит дій користувачів	Необхідні вбудовані правила розрахунку часу й логування
Нефункціональні	Безпека, автентифікація, контроль прав доступу	Визначає виділений Auth Service з шифруванням паролів bcrypt
Нефункціональні	Швидкість відгуку, пагінація, фільтрація черги заявок	Вимагає індексування полів бази даних та дебаунсу запитів

ПРАКТИЧНА ЧАСТИНА ДИПЛОМНОЇ РОБОТИ

Архітектура та стек технологій

Сервіси Backend
Python 3.12, FastAPI (асинхронний режим), Uvicorn

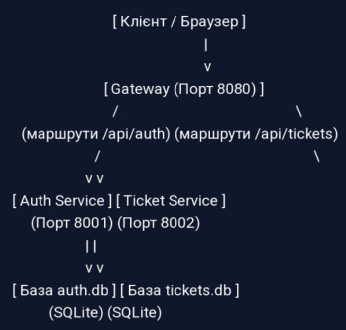
Бази даних
SQLite (для локальних тестів) / PostgreSQL (продакшн), SQLAlchemy ORM

Gateway (Маршрутизатор)
FastAPI Gateway proxy (локально) / Nginx Reverse Proxy (docker)

Frontend (SPA)
HTML5, Vanilla CSS3 (Premium Glassmorphism), JavaScript SPA

Безпека та токени
JWT (JSON Web Tokens), bcrypt для хешування паролів

СХЕМА КЛІЄНТ-СЕРВЕРНОЇ ВЗАЄМОДІЇ:



ПРАКТИЧНА ЧАСТИНА ДИПЛОМНОЇ РОБОТИ

Мережева специфікація контейнерів

Розподіл портів та призначення мікросервісів у Docker Compose (з диплому):

Назва сервісу	Внутрішній порт	Зовнішній порт	Функціональне призначення в системі
postgres	5432	5432	Реляційне збереження даних користувачів та заявок
auth_service	8000	8001	Автентифікація, авторизація, JWT та управління ролями (RBAC)
ticket_service	8000	8002	CRUD-операції із заявками, обробка коментарів, файлів та SLA
gateway	80	8080	Проксування запитів, єдина точка входу, роздача статики frontend

ПРАКТИЧНА ЧАСТИНА ДИПЛОМНОЇ РОБОТИ

Специфікація маршрутів REST API

Ключові кінцеві точки (Endpoints) розроблених мікросервісів:

Група API	Метод HTTP	Маршрут (Route)	Призначення запиту
Auth / Користувачі	POST	/api/auth/register	Реєстрація нового користувача в системі
Auth / Користувачі	POST	/api/auth/login	Автентифікація та отримання сесійного JWT-токена
Tickets (Заявки)	POST	/api/tickets	Створення нового звернення з пріоритетом (роль client)
Tickets (Заявки)	GET	/api/tickets	Отримання списку заявок (з фільтрацією та пагінацією)
Tickets (Заявки)	PATCH	/api/tickets/{id}	Оновлення статусу, пріоритету або виконавця тікета
Comments / Files	POST	/api/tickets/{id}/comments	Додавання нового коментаря в історію вирішення

ПРАКТИЧНА ЧАСТИНА ДИПЛОМНОЇ РОБОТИ

Модель даних та структури таблиць БД

Основні сутності бази даних та зв'язки між ними:

Сутність (Entity)	Ключові поля таблиці (SQL attributes)	Призначення таблиці
User (Користувач)	id, username, email, password_hash, role (admin/agent/client), created_at	Зберігання профілів та визначення ролей
Ticket (Заявка)	id, subject, description, priority, status, created_by, assigned_to, first_response_due_at, resolution_due_at	Облік тікетів, опис інцидентів та мітки SLA
Comment (Коментар)	id, ticket_id, author_id, text, created_at	Фіксація історії обговорення звернення

ПРАКТИЧНА ЧАСТИНА ДИПЛОМНОЇ РОБОТИ

Авторизація та безпека (Auth Service)

- **Автентифікація**

Користувач надає email та пароль. Сервіс звіряє хеш пароля з базою даних.

- **Шифрування bcrypt**

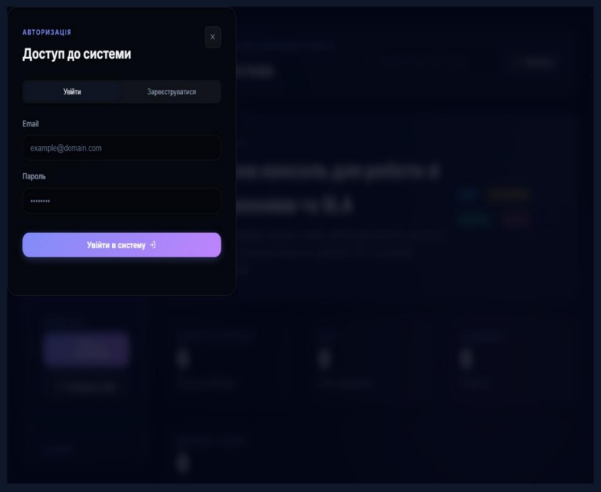
Паролі ніколи не зберігаються у відкритому вигляді. Використовується стійке солене хешування bcrypt.

- **Генерація токенів JWT**

Після перевірки даних генерується безсесійний JWT-токен, підписаний алгоритмом HS256, що містить ідентифікатор користувача, його роль та термін дії.

- **Безпечна взаємодія**

Браузер зберігає токен у sessionStorage та прикріплює його до заголовка Authorization (Bearer <TOKEN>) для захищених запитів API.



ПРАКТИЧНА ЧАСТИНА ДИПЛОМНОЇ РОБОТИ

Контроль регламентів обслуговування (SLA)

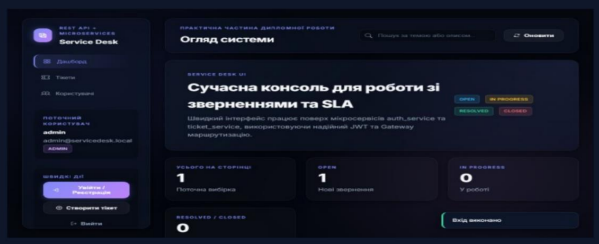
Автоматичний розрахунок термінів реакції та вирішення інцидентів (з диплому):

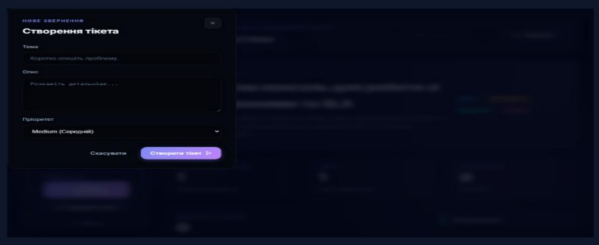
Пріоритет звернення	Час першої реакції	Час повного вирішення	Практичний опис для IT
low (Низький)	24 години	72 години	Планові завдання, загальні запитання
medium (Середній)	8 годин	24 години	Локальні збої, стандартні інциденти
high (Високий)	4 години	8 годин	Порушення роботи окремого відділу
critical (Критичний)	1 година	4 години	Повна зупинка бізнес-процесу

ПРАКТИЧНА ЧАСТИНА ДИПЛОМНОЇ РОБОТИ

Робоче місце IT -агента (Dashboard)

- **Зручний огляд метрик**
Дашборд відображає підсумкову статистику за всіма поточними зверненнями.
- **Контроль термінів SLA**
Динамічний аналіз черги попереджає про наближення дедлайнів реагування.
- **Швидка реєстрація**
Модальне вікно створення заявки з вибором теми, опису та рівня пріоритету.

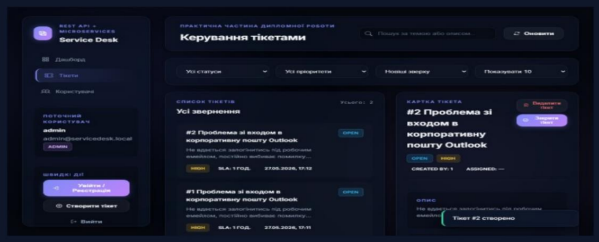


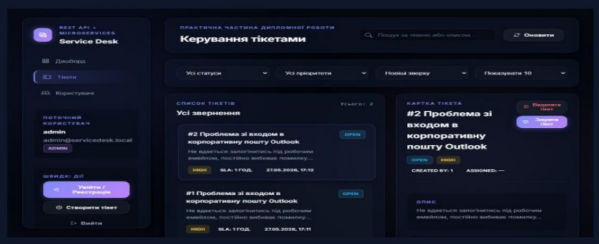


ПРАКТИЧНА ЧАСТИНА ДИПЛОМНОЇ РОБОТИ

Список та детальна картка тікета

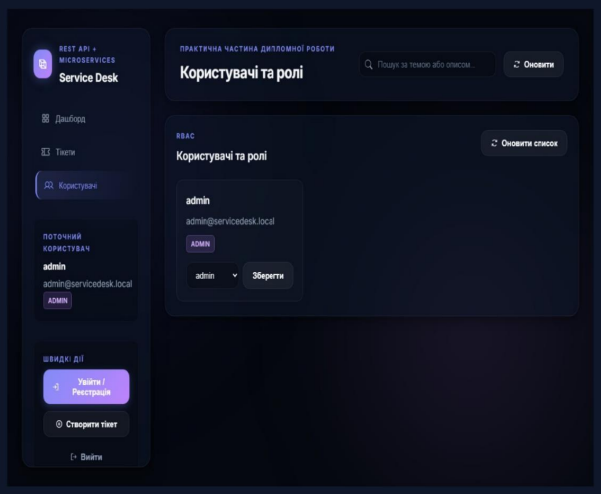
- **Фільтрація та пошук**
Впроваджено миттєвий пошук по тексту заявок та фільтрацію за статусом/пріоритетом.
- **Пагінація черги**
Зменшує навантаження на БД завдяки порційному завантаженню даних.
- **Спільне обговорення**
Можливість ведення внутрішньої переписки клієнт-агент у реальному часі.





Управління ролями користувачів (RBAC)

- **Адміністративний контроль**
  Розділ доступний виключно для користувачів з роллю admin.
- **Динамічне призначення прав**
  Зміна ролі (client/agent/admin) здійснюється безпосередньо в інтерфейсі картки користувача.
- **Консистентність даних**
  Усі зміни ролей миттєво відправляються на Auth Service та зберігаються в базі даних.



Оцінка результатів та висновки

ВИСНОВКИ:

— Розроблено та успішно протестовано працездатний прототип мікросервісної REST API системи Service Desk.

— Впровадження мікросервісів забезпечило високу відмовостійкість і незалежне розгортання окремих модулів.

— Вдалося автоматизувати контроль SLA-регламентів реакції та вирішення для підвищення якості обслуговування.

МЕТРИКИ НАВАНТАЖУВАЛЬНОГО ТЕСТУВАННЯ:

Операція API	К-сть запитів	Час відповіді
Авторизація	700	95 мс
Отримання списку	1000	120 мс
Створення заявки	500	180 мс
Додавання коментаря	500	160 мс